8-23-2017 1:30 PM

An Internet-Wide Analysis of Diffie-Hellman Key Exchange and X.509 Certificates in TLS

Kristen Dorey, *The University of Western Ontario*

Supervisor: Dr. Aleksander Essex, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

Recommended Citation

Dorey, Kristen, "An Internet-Wide Analysis of Diffie-Hellman Key Exchange and X.509 Certificates in TLS" (2017). *Electronic Thesis and Dissertation Repository*. 4792.
https://ir.lib.uwo.ca/etd/4792

Abstract

Transport Layer Security (TLS) is a mature cryptographic protocol, but has flexibility during implementation which can introduce exploitable flaws. New vulnerabilities are routinely discovered that affect the security of TLS implementations.

We discovered that discrete logarithm implementations have poor parameter validation, and we mathematically constructed a deniable backdoor to exploit this flaw in the finite field Diffie-Hellman key exchange. We described attack vectors an attacker could use to position this backdoor, and outlined a man-in-the-middle attack that exploits the backdoor to force Diffie-Hellman use during the TLS connection.

We conducted an Internet-wide survey of ephemeral finite field Diffie-Hellman (DHE) across TLS and STARTTLS, finding hundreds of potentially backdoored DHE parameters and partially recovering the private DHE key in some cases. Disclosures were made to companies using these parameters, resulting in a public security advisory and discussions with the CTO of a billion-dollar company.

We conducted a second Internet-wide survey investigating X.509 certificate name mismatch errors, finding approximately 70 million websites invalidated by these errors and additionally discovering over 1000 websites made inaccessible due to a combination of forced HTTPS and mismatch errors. We determined that name mismatch errors occur largely due to certificate mismanagement by web hosting and content delivery network companies. Further research into TLS implementations is necessary to encourage the use of more secure parameters.

Keywords: Transport Layer Security, discrete logarithm problem, Diffie-Hellman, small subgroup attack, X.509 certificate, name mismatch error

Acknowledgements

I want to thank my supervisor, Aleksander Essex, for his unfailingly positive support and for routinely teaching me something new about software. Thank you for taking a chance on me.

I also want to thank my family, friends, and Whisper Lab members for providing much appreciated support and fun in equal measures. You are all awesome.

Co-Authorship Statement

In this thesis, Section 2.4.4, Chapter 3, and Chapter 4 were adapted from Dorey et al.'s article "Indiscreet Logs: Diffie-Hellman Backdoors in TLS", which was co-authored by Nicholas Chang-Fong and Aleksander Essex.

Nicholas Chang-Fong and Aleksander Essex provided the initial idea for the article. Aleksander Essex additionally contributed to multiple areas: finding related work in Section 2.4.4, providing the idea behind and additionally conducting the testing in Sections 3.2-3.3, providing the mathematical concepts and constructions in Section 3.4, providing the information behind Section 4.3.1, providing the idea behind and additionally conducting the testing in Section 4.3.2, creating the attack described in Section 4.4, and contributing ideas in Sections 4.5-4.7.

Contents

List of Figures

List of Tables

List of Appendices

List of Abbreviations, Symbols, and Nomenclature

A1	A1 Telekom Austria
AES	Advanced Encryption Standard
AWS	Amazon Web Services
BCP	Banco de Crédito
CA	Certification/certificate authority
CDH	Computational Diffie-Hellman assumption
CDN	Content delivery network
CIRA	Canadian Internet Registration Authority
CN	Common name
CNA	CVE Numbering Authority
CNRS	Centre national de la recherche scientifique
CRT	Chinese remainder theorem
CVE	Common Vulnerabilities and Exposures, and the informal name for a CVE identifier
CVSS	Common Vulnerability Scoring System
CZDS	Centralized Zone Data Service
DDH	Decisional Diffie-Hellman assumption
DER	Deutsche Reisebüro

DHE	Finite Field Diffie-Hellman (ephemeral)
DL	Discrete logarithm
DLP	Discrete logarithm problem
DN	Distinguished name
DNS	Domain Name System
DROWN	Decrypting RSA using Obsolete and Weakened eNcryption
DSA	Digital Signature Algorithm
ECDHE	Elliptic Curve Diffie-Hellman (ephemeral)
ECDSA	Elliptic Curve Digital Signature Algorithm
FQDN	Fully Qualified Domain Name
GCM	Galois/Counter Mode
GNFS	Generalized number field sieve
GPG	GNU Privacy Guard
HSTS	HTTP Strict Transport Security
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure or HTTP over SSL/TLS
IETF	Internet Engineering Task Force
IMAP	Internet Message Access Protocol
IMAPS	IMAP Secure or IMAP over SSL/TLS
IP address	Internet Protocol address
IPSec	Internet Protocol Security
IPv4	Internet Protocol version 4
LAN	Local area network
MAC	Message authentication code

MITM	Man-in-the-middle
MODP	Modular Exponential
NIST	National Institute of Standards and Technology
NS	Nederlandse Spoorwegen
NVD	U.S. National Vulnerability Database
OSI	Open Systems Interconnection
PGP	Pretty Good Privacy
PKI	Public-key infrastructure
POODLE	Padding Oracle On Downgraded Legacy Encryption
POP	Post Office Protocol
POP3	Post Office Protocol (version 3)
POP3S	POP3 Secure or POP3 over SSL/TLS
PRF	Pseudorandom function
PRNG	Pseudorandom number generator
RFC	Request for Comments
RSA	Rivest-Shamir-Adleman cryptosystem
SAN	Subject alternative name
SCU	Santa Clara University
SHA-2	Secure Hash Algorithm 2
SMTP	Simple Mail Transfer Protocol
SMTPS	SMTP Secure or SMTP over SSL/TLS
SNI	Server Name Indication
SSH	Secure Shell
SSL	Secure Sockets Layer

STARTTLS	Extension to upgrade existing connection to be secure over TLS
TCP	Transmission Control Protocol
TLD	Top-Level Domain
TLS	Transport Layer Security
UNED	Universidad Nacional de Educación a Distancia
UPS	United Parcel Service
VPN	Virtual Private Network
WPA2	Wi-Fi Protected Access II
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

Simply put, if you use the Internet, you have used Transport Layer Security (TLS). The green padlock icon (Figure 1.1) displayed in the browser on your desktop or cellphone shows that TLS is used by that website. Most users unknowingly entrust TLS to secure services such as online banking, email, and internet voting – without it, an attacker can see your banking information, obtain your email passwords, or see who you voted for in an online election. New vulnerabilities such as Logjam [8] and DROWN (Decrypting RSA using Obsolete and Weakened eNcryption) [12] are routinely discovered that could undermine the security of those systems.

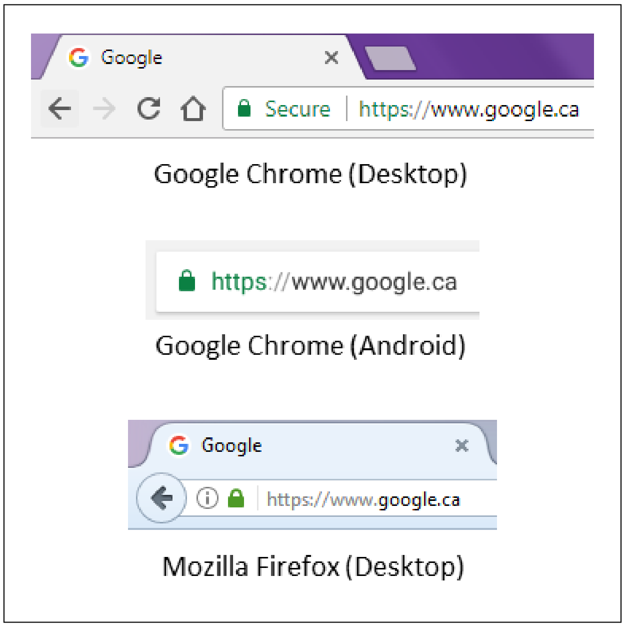


Figure 1.1: **The Green Padlock Icon.** Examples of the green padlock for Google Chrome and Mozilla Firefox.

Comprehensive checks of vulnerable systems can be done with Internet-wide data sets, which also provide insight into the less travelled corners of the Internet. In the last few years, Internet-wide scanning has become easier and more popular with application-layer scanners such as ZGrab [36]. With this wealth of data, there are aspects of current TLS deployment that remain uninvestigated. This thesis demonstrates a new vulnerability in the implementation of TLS and additionally presents a survey of a well-known TLS misconfiguration.

1.1 Motivation

TLS is mature and makes excellent use of cryptography to provide security, but implementations of TLS are open to interpretation which can introduce vulnerabilities [8, 12, 81, 38]. This work aims to make TLS implementations more secure: we demonstrate that it only takes one weak spot for an entire implementation to become vulnerable, and we encourage implementations to follow best practices even when an attack is not immediately evident.

1.2 Contributions

This thesis makes five contributions to the study of TLS implementations, of which the first four were published at the 2017 Network and Distributed System Security Symposium [35]:

1. We outline a method for mathematically constructing a backdoor that remains deniable while exploiting poor parameter validation in discrete logarithm implementations.

2. We conducted an Internet-wide survey of ephemeral Diffie-Hellman (DHE) support, uncovering hundreds of TLS- and STARTTLS-enabled web and mail servers using composite moduli. These potentially backdoored parameters were found across a range of protocols – including HTTPS, SMTP, SMTPS, IMAPS, and POP3S – and spanned over 30 countries and a diverse set of organizations. In some cases, we were able to recover large portions of the private DHE key. We additionally found 1.6 million servers offering non-safe prime groups of unknown order.

3. We discuss how TLS 1.2 and earlier is vulnerable to a man-in-the-middle attack, where an attacker that can exploit backdoored parameters can force a DHE cipher suite to be negotiated as long as both parties support it. We present several possible attack vectors to deliver these malicious parameters: directly attacking the server or TLS endpoint, or by attacking the software upstream.

4. We disclosed the potentially backdoored parameters to 17 companies, resulting in a public security advisory (CVE-2016-5774) and conference calls with the CTO of a billion-dollar company. The organizations we spoke to declined to explain how composite moduli came to be used in their DHE configurations.

5. We conducted an Internet-wide survey of X.509 certificates invalidated by name mismatch errors, uncovering approximately 70 million websites with this error. We categorized these errors and determined that web hosting or content delivery network (CDN) companies were the most common cause. We additionally found over 1000 websites with this error that forced HTTPS use, making their websites inaccessible.

1.3 Organization of Thesis

The remainder of this thesis is organized into five chapters:

- **Chapter 2** details the Transport Layer Security (TLS) protocol and its uses. It further discusses two aspects of TLS: the finite field Diffie-Hellman key exchange in terms of its purpose, cryptographic operations, and methodology; and X.509 certificates in terms of their purpose, trust hierarchy, structure, and invalidating errors.

- **Chapter 3** outlines and demonstrates the lack of parameter validation found in discrete logarithm implementations, and explains the construction of a backdoor that exploits this weakness.

- **Chapter 4** describes an Internet-wide survey into potentially backdoored parameters across TLS and STARTTLS; presents a man-in-the-middle attack to force DHE use, which requires an attacker to first position the backdoor through attack vectors we describe; and details vulnerability disclosures.

- **Chapter 5** describes an Internet-wide survey into certificate-invalidating name mismatch errors, and outlines the impact of these errors on websites that force HTTPS use.

- **Chapter 6** discusses the declining support for finite field Diffie-Hellman due to the combination of our work and others, and outlines potential future work in the area of name mismatch errors.

Chapter 2

Background

A version of § 2.4.4 has been published as part of [35].

2.1 Network Security

When considering the security of communications over a network, the network can be conceptualized as layers that are secured separately [80, 67]. There are seven layers defined by the Open Systems Interconnection (OSI) model: *physical* (the lowest layer), *data link*, *network*, *transport*, *session*, *presentation*, and *application*. Security requirements differ across layers, meaning only some layers have security protocols and those protocols vary. For example, in the data link layer, a wireless local area network (LAN) such as Wi-Fi can be secured through the Wi-Fi Protected Access II (WPA2) protocol. In the network layer, a Virtual Private Network (VPN) can be secured through Internet Protocol Security (IPSec). In the transport layer, which provides end-to-end communication between applications on network-connected hosts, a protocol such as the Transmission Control Protocol (TCP) can be secured with confidentiality, integrity, and authenticity through Transport Layer Security (TLS).

2.2 Transport Layer Security

In this section, we discuss the goals, subprotocols, and applications of Transport Layer Security (TLS) protocol.

2.2.1 What is Transport Layer Security?

The Transport Layer Security (TLS) protocol is a cryptographic protocol used to secure communication at the transport layer of a network. TLS 1.2 [33] was finalized in 2008, and TLS

1.3 [72] is currently a draft. TLS provides a security and proficiency upgrade to the Secure Sockets Layer (SSL), which had its final version SSL 3.0 deprecated in 2015 [15] after critical security vulnerabilities were discovered. For example, the fix for the POODLE attack (Padding Oracle On Downgraded Legacy Encryption) [64] required extensions, which are only possible in TLS 1.0 and above. Despite this, SSL 3.0 is still used in a few HTTPS connections today [52].

Goals of TLS. RFC 5426 [33] specifies four goals for TLS in order of importance: cryptographic security for connections, interoperability between different applications, extensibility for future expansions, and relative efficiency for cryptographic operations. Expanding the first goal, TLS is intended to secure communicating applications by supplying confidentiality, integrity, and authenticity to the connection. Confidentiality is provided through encryption, and protects against eavesdropping and also data theft from either server or client. Integrity is provided through message authentication codes, and protects against data, memory, and message traffic modification. Finally, authenticity is provided through digital signatures, certificates, and public key cryptography. It protects against impersonation and data forgery [79].

Uses of TLS. TLS is placed above the transport layer, but does not fit neatly into an OSI model layer. It can therefore be used with any application protocol. Common application protocols used include Hypertext Transfer Protocol (HTTP) for communicating on the World Wide Web; Simple Mail Transfer Protocol (SMTP) for transmitting email; Internet Message Access Protocol (IMAP) and Post Office Protocol (POP) for retrieving email; and Extensible Messaging and Presence Protocol (XMPP) for instant messaging [78]. These protocols are discussed further in § 2.2.4. The freedom to choose application protocols means that their TLS implementations are not specified, enabling different interpretations and providing openings for vulnerable configurations. The remainder of § 2.2 discusses the inner workings of TLS to provide context for the vulnerability discussions in Chapter 4 and Chapter 5.

Layers of TLS. There are two layers to TLS: the TLS Record Protocol, and the TLS Handshake Protocol which also contains subprotocols. The first layer is the TLS Record Protocol, placed above the transport layer. The TLS Record Protocol takes higher-layer data that needs to be transmitted, divides it into blocks, and potentially compresses the data. A message authentication code (MAC – not to be confused with a Media Access Control address) is then added to the record, followed by encrypting the data based on the previously negotiated cipher and sending the final product to the transport layer [33].

The second layer of TLS is the TLS Handshake Protocol, placed above the Record Protocol.

It has three subprotocols: the Change Cipher Spec Protocol, used for changing previously negotiated ciphers during the handshake; the Alert Protocol, used for sending warning messages as information or fatal messages to terminate the connection; and the Handshake Protocol. The Handshake Protocol is used by a client, such as a browser, and a server to determine cryptographic keys that enable secure communication between the parties. For simplicity, further references to the Handshake Protocol refer to the subprotocol.

2.2.2 TLS Handshake Protocol

Handshake Types. In general, there are three types of the TLS Handshake: abbreviated handshake, and full handshake with or without client authentication [76]. The abbreviated handshake is the most common method, and is done when resuming a session created from a previous full handshake. The main advantage to abbreviating a handshake using already negotiated security parameters is the computational cost reduction. The full handshake can be seen with client authentication if the server has been successfully authenticated. However, client authentication is not often done; instead, a client is usually authenticated through a username and password [44]. This section expands upon the full handshake with optional client authentication for completeness.

Overview of Full Handshake. The full TLS Handshake consists of `Hello` messages; certificate requests, receipts, and verifications; key exchanges; and `Finished` messages [33]. The handshake sequence, illustrated in Figure 2.1, consists of a maximum of eleven messages in specific order: client hello, server hello, server certificate, server key exchange, certificate request, server hello done, client certificate, client key exchange, certificate verify, client finished, and server finished. The Change Cipher Spec messages are part of the Change Cipher Spec Protocol, not the Handshake Protocol, but their placement is related to the handshake sequence as explained in § 2.2.3.

2.2.3 TLS Handshake Messages

(1) **Client Hello.** To initiate a TLS connection, the client sends a `ClientHello` to the server. We focus on four important aspects of the `ClientHello`:

A. **List of supported cipher suites.** A cipher suite in TLS 1.2 and below, seen in Figure 2.2, defines the algorithms used in the rest of the handshake:

i. **Encryption.** A cipher algorithm and its mode of operation are used for encryption. A common choice is the Advanced Encryption Standard (AES) operating

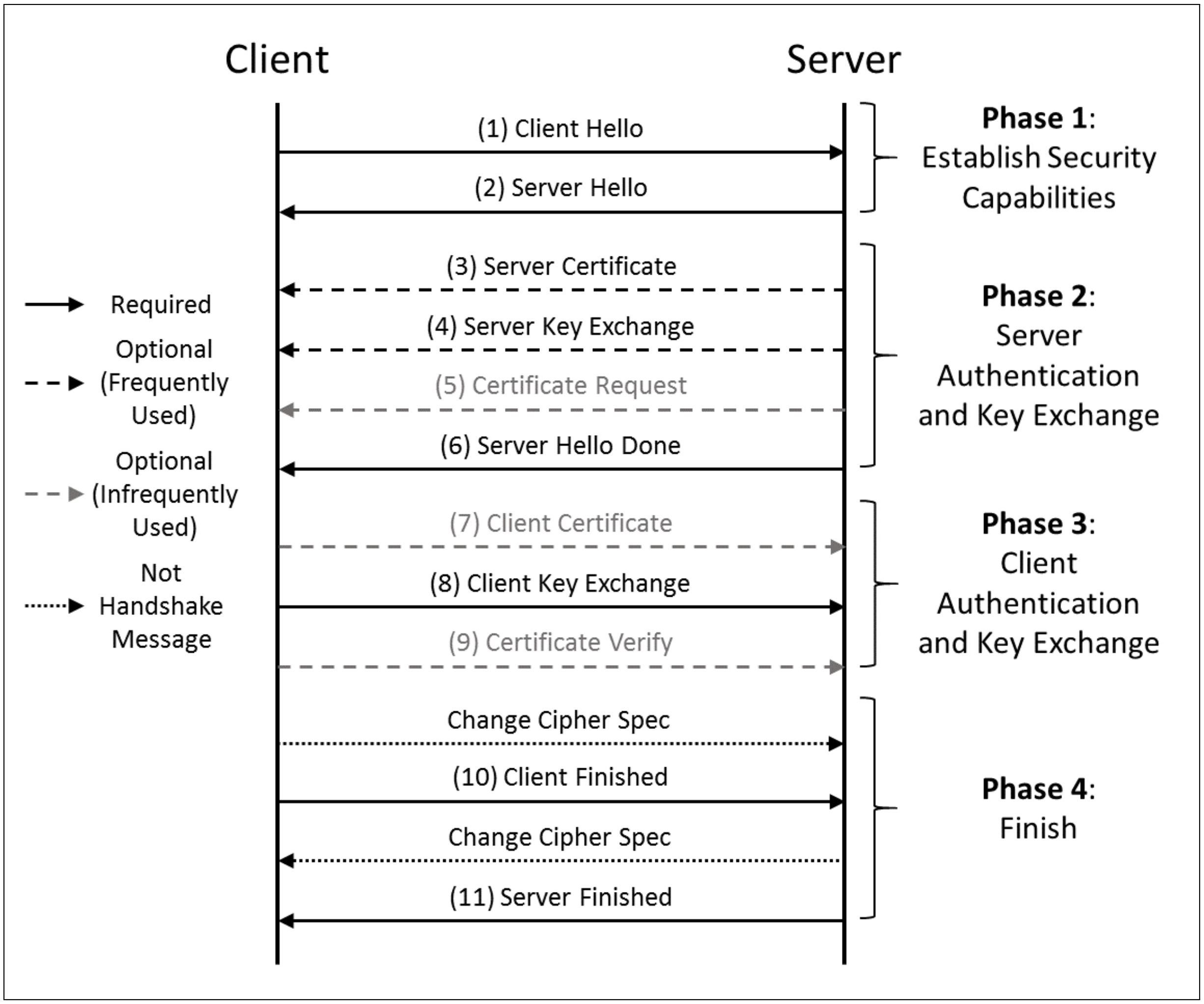


Figure 2.1: **The TLS Handshake.** The TLS Handshake Protocol can be broken into four phases [79].

by Galois/Counter Mode (GCM), for security and efficiency respectively.

ii. **Key Exchange.** A client and server exchange keys that are later used to derive the encryption and message authentication code (MAC) keys for the connection. Key exchange is described later in this section.

iii. **Authentication/Signature.** Public keys exchanged may be signed to prove their authenticity, depending on the key exchange method. Signature verification is confirmed through certificates.

iv. **Hash Function.** A hash function is used when creating the MAC. A common choice is the Secure Hash Algorithm 2 (SHA-2) for security. In TLS 1.2, the hash function can also be used in the pseudorandom function (PRF) for key derivation.

B. **Random bytes.** The included random bytes (ClientHello.random) are used later during key generation to produce unique encryption and authentication keys for the TLS connection. Some of these bytes are epoch time – the number of seconds since January 1, 1970. Unique keys are needed to prevent replay attacks, where an attacker saves data from a previous connection and resends it to one party, producing a valid connection.

C. **TLS version.** The client includes its desired TLS version in the ClientHello, which is the highest version it supports.

D. **Session identifier.** The session identifier is used if reusing security parameters from a previous session (see abbreviated handshake from § 2.2.2).

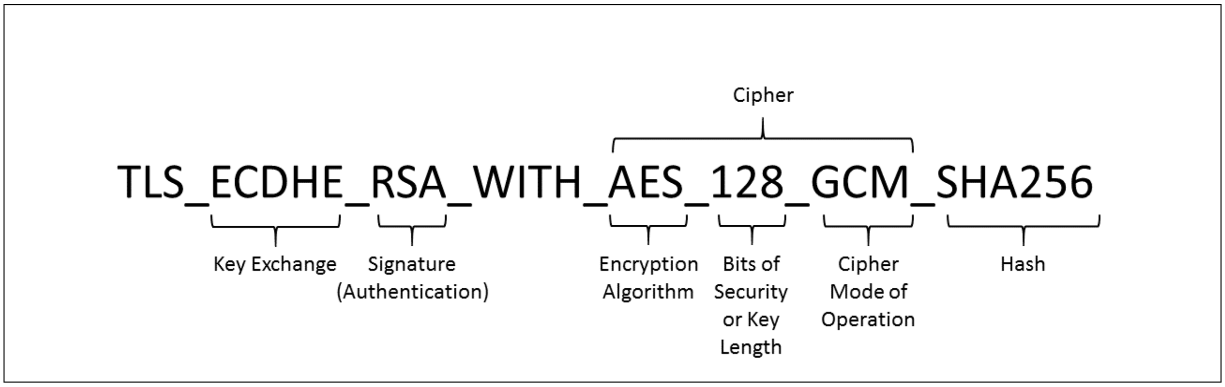

Cipher
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
Key Exchange
Signature (Authentication)
Encryption Algorithm
Bits of Security or Key Length
Cipher Mode of Operation
Hash

Figure 2.2: **A TLS cipher suite.** An example cipher suite supported by Chrome 57.

(2) **Server Hello.** After the ClientHello, the server must respond with its ServerHello. It is structured similar to the ClientHello; for example, it also contains random bytes ServerHello.random. However, in general the ServerHello contains the server's selections rather than options. For example, the server usually picks its most preferred cipher suite that the client also supports.

(3) **Server Certificate.** The message following the ServerHello is the server's digital certificate(s), which is the first step in authenticating the server to the client. Server authentication is explained later in the ServerKeyExchange since it relies on both the server's certificate and key exchange. Except in rare cases, the server must always send at least one public key certificate to the client, where multiple certificates form a chain. More specifically, a server's Certificate message is required when using any key exchange method defined in TLS 1.2, except for one which is deprecated in TLS 1.3 [72]. For all versions of TLS, X.509 version 3 (v3) certificates [28] are the default, although experimental methods exist such as OpenPGP certificates [61] derived from Pretty Good

Privacy (PGP). X.509 certificates are explained further in § 2.5. Among other fields, these certificates contain the server's public key, which is used for server authentication through encryption or signature verification depending on the key exchange algorithm.

(4) **Server Key Exchange.** A server's certificate chain, or the `ServerHello` if no `Certificate` message was sent, is typically followed by the `ServerKeyExchange`. This message is required if the client needs additional information to generate the premaster secret. The premaster secret is explained later in the `ClientKeyExchange`.

Key exchange finishes server authentication that started with the server's certificate. Key exchange algorithms requiring `ServerKeyExchange` include ephemeral finite field Diffie-Hellman (DHE) and ephemeral Elliptic Curve Diffie-Hellman (ECDHE) [23], where ephemeral means the DHE/ECDHE keys are used only once. Using Rivest-Shamir-Adleman (RSA) or fixed/static Diffie-Hellman for key exchange does not require a `ServerKeyExchange` message since the client obtains the public parameters needed for premaster secret generation from the server's certificate [67]. For RSA, server authentication is finished by encrypting the premaster secret with the server's public key from its certificate – the server is authenticated since it must use the corresponding private key to decrypt.

Definition. (*Ephemeral.*) A key is ephemeral if it is used only once.

If a `ServerKeyExchange` message is sent, it contains the DHE or ECDHE parameters (see § 2.4) along with a signature of those parameters. The server creates the signature by hashing the parameters with the `ClientHello` and `ServerHello` randoms, then encrypting the hash with the private key that matches the public key on the server's certificate. The client uses that public key, previously obtained through the server's certificate, to verify the signature – the server is authenticated since it must have used the corresponding private key to sign the parameters.

(5) **Certificate Request.** This step is the first in client authentication, which is not frequently done in the TLS handshake. Client authentication is different than server authentication; it still depends upon certificates but signs previously exchanged messages instead of key exchange parameters. Client authentication is expanded upon in the client's `Certificate` and `CertificateVerify`. A server previously authenticated with its certificate and key exchange can send a `CertificateRequest` to the client after the `ServerKeyExchange`. If there was no `ServerKeyExchange`, this message is sent after the server's `Certificate`. The `CertificateRequest` contains the types of keys the client's certificate can contain, among other information.

(6) **Server Hello Done.** A server must send an empty `ServerHelloDone` message to inform the client that it has sent all of its handshake messages. It does not respond again until it is time for its `Finished` message. After receiving the `ServerHelloDone`, the client should check the acceptability of the `ServerHello` message and the validity of the server's certificate chain if one exists.

(7) **Client Certificate.** This is the second step in client authentication, and therefore it is not often done. If the server has previously sent a `CertificateRequest`, the client must respond with a `Certificate` message. A server may choose to continue the handshake even if the `Certificate` message contains no certificates. The client's `Certificate` message follows the same format as the server's `Certificate` message – one of its fields is the client's public key, used in client authentication. This message must also be compatible with the specifications outlined in the server's `CertificateRequest`.

(8) **Client Key Exchange.** A client's certificate chain, or the `ServerHelloDone` if no `Certificate` message was sent, must be followed by the `ClientKeyExchange`. This message ensures both parties have the premaster secret `pre_master_secret`, although the exact message depends on the key exchange algorithm. For example, with DHE or ECDHE key exchange, the client sends its public DHE or ECDHE parameters so that the server can compute the premaster secret. In RSA key exchange, the client computes the premaster secret itself and sends it to the server after encrypting it with the public key from the server's certificate.

The premaster secret with the addition of the random bytes from the `ClientHello` and `ServerHello` are used to generate the 48-byte master secret [33]:

```
master_secret = PRF(pre_master_secret, "master secret",
          ClientHello.random + ServerHello.random)
```

The master secret `master_secret` then generates the key block [33]:

```
key_block = PRF(master_secret, "key expansion",
    ServerHello.random + ClientHello.random);
```

The key block is then split into a client encryption key, server encryption key, client message authentication code (MAC) key, and server MAC key.

(9) **Certificate Verify.** This is the last step in client authentication, and as such it is not often done. If a client has sent a certificate with signing capabilities, it must complete its authentication to the server with a `CertificateVerify` message following the `ClientKeyExchange`. Therefore this message must always be sent except when fixed/static Diffie-Hellman was used for key exchange. The `CertificateVerify` message is a signed hash of previously exchanged handshake messages from `ClientHello` to before `CertificateVerify`. Similar to server authentication, the server uses the public key previously obtained through the client's certificate to verify the signature – the client is authenticated since it must have used the corresponding private key to sign the messages. After this message is received by the server, the parties are ready to exchange `Finished` messages.

(10) **Client Finished.** Before the client sends its `Finished` message, it sends a `ChangeCipherSpec` message that is part of the Change Cipher Spec Protocol. This message indicates that the client has enough information use an encrypted connection with its generated keys. The client then sends its `Finished` message which is secured with the previously negotiated algorithms. The `Finished` message is a hash of all previously exchanged messages from the `ClientHello` to before the client's `Finished` message. It verifies that the key exchange and authentication(s) were done properly.

(11) **Server Finished.** After verifying the client's `Finished` message, the server sends a `ChangeCipherSpec` message similar to the client. It then sends its own `Finished` message, which is of the same format as the client's except that the hash also includes the client's `Finished` message. After the client has verified the server's message, the two parties can now exchange application data.

2.2.4 Applications of TLS

As mentioned in § 2.2.1, common application protocols protected by TLS include the Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), Internet Message Access Protocol (IMAP), and Post Office Protocol (POP). The application of TLS with these protocols is explained further in this section.

HTTPS. The Hypertext Transfer Protocol (HTTP) is the application protocol used for communication via the Internet. HTTPS is the implementation of HTTP and either SSL or TLS. As such, HTTPS is defined as HTTP Secure, HTTP over SSL, or HTTP over TLS [71]. HTTPS has been used by web browsers such as Google Chrome (i.e. Chrome) and Mozilla Firefox (i.e. Firefox) for years to secure communication between the browser and web server.

An example use of HTTPS is to secure password transmission on login pages, and in fact some websites drop the HTTPS connection for HTTP after this user authentication. More generally, many websites simply do not use HTTPS. Since HTTP communicates information over unencrypted channels, an attacker can easily view information passed along the connection. By contrast, an attacker wanting information from a site secured with HTTPS needs to conduct a man-in-the-middle (MITM) attack to gain information about or modify the connection. One such attack is HTTPS stripping [60], which can be prevented by forcing HTTPS-only use through HTTP Strict Transport Security (HSTS).

HSTS. HTTP Strict Transport Security (HSTS) [50] is a mechanism introduced recently that allows websites to specify that they should only be accessed through HTTPS. Two important attacks that HSTS prevents are HTTPS stripping [60] and attacks using invalid certificates, by forcing HTTPS use and disabling user circumvention respectively. HSTS can be implemented in two ways: setting the `Strict-Transport-Security` header in the HTTP response (activates when the website is accessed over HTTPS), or submitting the website to a HSTS preload list. Chrome has a HSTS preload list, and many major browsers such as Firefox have HSTS preload lists adapted from it [76].

SMTP and SMTPS. The Simple Mail Transfer Protocol (SMTP) is used for transmitting email from the email sender to its final server destination. SMTP can use TLS directly, called SMTP Secure or SMTP over SSL/TLS (SMTPS), or through the STARTTLS extension [51]. STARTTLS upgrades an existing connection to be secure over TLS, and so works over the same port as the unsecured protocol. By contrast, the protocol over TLS is used over a separate encrypted port.

IMAP/S and POP3/S. Both the Internet Message Access Protocol (IMAP) and Post Office Protocol version 3 (POP3) are used by email clients to retrieve email from an email server [79]. Similar to SMTP, either can use TLS directly (IMAPS and POP3S) or through the STARTTLS extension.

2.3 Secure Shell

Similar to TLS, the Secure Shell (SSH) protocol is used to secure communication at the transport layer of a network. Its newest version is outlined in Internet Engineering Task Force (IETF) Request for Comments (RFC) 4250 to 4256. However among other differences, TLS is commonly used to secure HTTP while SSH is known for securing remote logins. Most importantly for this discussion, SSH can use Diffie-Hellman for key exchange [87] similar to TLS. A major implementation of SSH is OpenSSH,[1] which we discuss briefly in § 4.4.2.

2.4 Diffie-Hellman

The current primary key exchange algorithm used in TLS is ephemeral Diffie-Hellman over finite fields (DHE) or elliptic curves (ECDHE). We say Diffie-Hellman is used over finite fields to distinguish it from elliptic curves; this terminology is widely used [8, 13, 42, 45]. At the time of writing, telemetry data shows that three key exchange methods are used in TLS handshakes: ECDHE accounts for 90-92%, RSA for 8-9%, and DHE for 0.01-1% [65]. Despite this, we show in § 4.2.2 that DHE is still widely supported; for example in HTTPS, DHE cipher suites were supported by 25% of servers. This section discusses the need for key exchange algorithms, the cryptography behind finite field Diffie-Hellman key exchange, and the related work in this area.

2.4.1 Public-Key Key Exchange

In § 2.2.3, the concept of key exchange was introduced as a step in the TLS handshake. It was assumed in that section that if a client and server wanted to communicate, they could securely transmit the keys needed to do so. In reality, during the TLS handshake the client and server are communicating over an insecure network, yet need a secure way to transfer keys.

The two types of key cryptography used in TLS are symmetric-key cryptography and public-key cryptography (also known as asymmetric-key cryptography). Symmetric-key cryptography is frequently used in encryption, where it uses the same key for encrypting plaintext (i.e. unencrypted information) and decrypting ciphertext (i.e. encrypted information). In TLS, symmetric keys are used for encryption/decryption and MACs; for example, the AES encryption scheme uses symmetric-key cryptography. Unfortunately, symmetric-key cryptography does not solve the problem of first having to securely communicate the key between parties, which is why in TLS symmetric keys are only used internally by the client and server.

The secure key transfer problem of symmetric-key cryptography is why public-key cryptography was invented. Before public-key cryptography, transferring keys securely was done by physical methods such as face-to-face meetings. This method had its own problems such as potential key loss or tampering en route. A secure key transfer method was needed, one which

[1] `https://www.openssh.com/`

allowed two parties to compute a shared secret such that no observer of the transfer could recover the shared secret. Each party has a different key pair: a public key (able to be exchanged over an insecure channel) and a private key (kept secret).

Definition. (*Key pair.*) A key pair, used in public-key key exchange, consists of a public key that can be exchanged over an insecure channel and a private key that is known only to the party who generated it.

The shared secret is computed separately by both parties using their own private key and the other's public key. Diffie and Hellman were the first to publicly propose such a method [34], and it was considered a major advance in secure communication since entities who had never met could now communicate securely. Their public-key key exchange protocol is called the Diffie-Hellman key exchange. The cryptography behind the Diffie-Hellman key exchange is discussed in § 2.4.2, and the actual protocol is discussed in § 2.4.3.

2.4.2 Cryptography

The cryptography behind the finite field Diffie-Hellman key exchange is discussed in this section.

Groups. A group $\{\mathbb{G}, *\}$ is a set of elements $\mathbb{G}$ such that a pair of elements (a,b) can be combined to form another element $(a*b)$ through a binary operation, $*$, such as addition or multiplication [63]. For the purposes of this discussion, multiplication is the most relevant binary operation. The group $\mathbb{G}$ needs to have four properties:

- **Closure.** For any two elements a,b in $\mathbb{G}$, $\mathbb{G}$ must also contain the combined element $(a*b)$.

- **Associative.** For any three elements a,b,c in $\mathbb{G}$, combining two of the elements with the remaining element should always produce the same result. For example, $a*(b*c) = (a*b)*c$.

- **Existence of Identity Element.** $\mathbb{G}$ includes an identity element e, which for multiplication of real numbers is 1. When combining the identity element with any other element, it always equals the second element. For example, $a*e = e*a = a$.

- **Existence of Inverse Element.** Each element a in $\mathbb{G}$ has a corresponding inverse element a^{-1}. When the two are combined, the result is the identity element e. For example, $a*a^{-1} = a^{-1}*a = e$.

The group $\mathbb{G}$ is considered an abelian group if it has an additional property:

- **Commutative.** For any two elements a, b in $\mathbb{G}$, combining the two elements in a different order should not change the result. For example, $a * b = b * a$.

Abelian groups are important to the definition of a field, where fields are fundamental to discrete logarithms and therefore Diffie-Hellman. A field F is a set of elements with binary operations of addition – under which F forms an abelian group – and multiplication, among other properties. For the purposes of this discussion, multiplication is the most relevant binary operation; under multiplication, the non-zero elements of the field form an abelian group [79]. A finite field possesses an order equal to the finite number of elements in it, so finite fields are more useful in cryptography than infinite fields. This order must equal p^n, where n is the positive integer that a prime p is raised to. This discussion is restricted to the case of $n = 1$, or a finite field with order p. This type of field is the set of integers $\mathbb{Z}_p = \{0, 1, ..., p-1\}$, where the elements can be multiplied modulo p. A similar field can be defined for other primes; for example, with prime q there exists $\mathbb{Z}_q = \{0, 1, ..., q-1\}$. All integers in $\mathbb{Z}_p$ except 0 are relatively prime to p, since the only common positive integer factor between p and each integer is 1. This property means that a multiplicative inverse exists for every integer in $\mathbb{Z}_p$ except 0. This set of invertible elements is $\mathbb{Z}_p^* = \{1, ..., p-1\}$, also called the multiplicative group of numbers modulo p. It is used to define a cyclic group $\mathbb{G}_q$.

Cyclic Groups. A cyclic group is an abelian group $\mathbb{G}$ where there exists an element g in $\mathbb{G}$, i.e. $g \in \mathbb{G}$, such that for all elements $a \in \mathbb{G}$ there exists an integer i such that $g^i = a$ where $g^i = g * g * ... * g$ i-times. The element g is called a generator because repeated applications of the binary operation to g generate the set of elements. The finite cyclic group $\mathbb{G}_q$ of order q is a subgroup of $\mathbb{Z}_p^*$, meaning it has only some of the elements from $\mathbb{Z}_p^*$. If q is prime, then all elements of $\mathbb{G}_q$ are generators. In general, $p = rq + 1$ where r is an integer and q is also prime. If p is a safe prime, this means that $r = 2$ so $p = 2q + 1$.

Definition. (*Safe prime.*) A safe prime, p_s, is a prime of the form $p_s = 2q + 1$, where q is also prime.

Definition. (*Safe prime group.*) A safe prime group, $\mathbb{G}_q$, is the q-order subgroup of $\mathbb{Z}_{p_s}^*$ (the multiplicative group of numbers modulo p_s), where p_s is a safe prime of the form $p_s = 2q + 1$.

Definition. (*Non-safe prime.*) A non-safe prime, p_n, is any prime that is not a safe prime.

Definition. (*Non-safe prime group.*) A non-safe prime group, $\mathbb{G}_q$, is the q-order subgroup of $\mathbb{Z}_{p_n}^*$ (the multiplicative group of numbers modulo p_n), where p_n is a non-safe prime.

Definition. (*Composite.*) A composite number, n, is any positive integer that is not prime.

The elements in $\mathbb{G}_q$ are generated by the generator g, which is also a group element. Not every group element is a generator; an element is only a generator if using the binary operation, $*$, on itself repeatedly generates all group elements.

Definition. (*Generator.*) A generator, g, generates the q-order subgroup $\mathbb{G}_q$ if the subgroup $\mathbb{G}_q = \{g, g^2, g^3, ..., g^q\}$, where $g^q = g^0 = 1$.

The concepts of generators, moduli, and $\mathbb{G}_q$ are used to shape the discrete logarithm problem.

Discrete Logarithms over Finite Fields. The discrete logarithm (DL) of an element a of $\mathbb{G}_q$ is defined as k where $a = g^k \bmod p$. The discrete logarithm problem (DLP) is therefore attempting to solve for k, given a modulus p and generator g of $\mathbb{G}_q$ which has an element a and order q. The DLP is computationally hard when the order q is large and is not smooth, meaning it cannot be factored into smaller primes.

Definition. (*Discrete logarithm problem.*) The discrete logarithm problem (DLP) involves attempting to solve for k, where $a = g^k \bmod p$ for a prime, p, and a generator, g, of the q-order subgroup $\mathbb{G}_q$ which has an element, a.

Definition. (*Smooth number.*) A b-smooth number, sn, is an integer that can be factored into a sequence of primes such that $sn = p_1 p_2 ... p_n$, where $p_i \leq b$ for some bound b. Informally, the term "smooth" number is used to describe a "small" b. In this thesis, we mean b is small enough such that solving the discrete logarithm in subgroups of order $p_i \leq b$ is efficient.

The current recommended key lengths by the National Institute of Standards and Technology (NIST) are $|p| \geq 2048$ bits and $|q| \geq 224$ bits [14]. The hardness of the DLP makes it the basis for DL implementations such as the Diffie-Hellman key exchange.

2.4.3 Diffie-Hellman Key Exchange

As mentioned at the beginning of § 2.4 and in § 2.4.1, ephemeral finite field Diffie-Hellman (DHE) is one of the public-key algorithms used in the key exchange portion of TLS. It makes use of the discrete logarithm problem (DLP) to calculate a shared secret between two parties communicating over a public authenticated channel [67]. If the DLP is sufficiently hard, then the Diffie-Hellman key exchange is theoretically secure.

Attacker Definitions. We have used the term "attacker" briefly, but the remainder of this chapter along with Chapters 3 and 4 require more specific definitions of "attacker". We take three attacker definitions used in cryptography:

- **Eve**: Eve is a passive attacker/eavesdropper, who is able to listen to communicated messages but is unable to change them;
- **Mallory**: Mallory is a malicious and active attacker who can change communicated messages;
- **Heidi**: Heidi is a malicious designer of cryptography parameters [42]. We additionally use Heidi to choose attack targets for installing her designed parameters.

These attacker names are used throughout this work.

Finite Field Diffie-Hellman Key Exchange. The finite field Diffie-Hellman key exchange is outlined in Figure 2.3. It starts with two users Alice (A) and Bob (B), who in TLS would be the client and server. An eavesdropper, Eve (E), can see any public parameters communicated between Alice and Bob. Alice and Bob agree upon a generator g of $\mathbb{G}_q$ and a modulus p, where their choices should ensure that the DLP is hard. These choices are called Diffie-Hellman domain parameters.

Definition. (*Diffie-Hellman domain parameters.*) Diffie-Hellman domain parameters (or simply DHE parameters) are modulus p (should be prime) and generator g of a q-order subgroup $\mathbb{G}_q$.

We use the notation $a \xleftarrow{\$} S$ to denote a value a sampled uniformly at random from set S. The parties independently choose a random integer from $\mathbb{Z}_q$, i.e. $k_a \xleftarrow{\$} \mathbb{Z}_q$ and $k_b \xleftarrow{\$} \mathbb{Z}_q$ for Alice and Bob respectively, where k_a is known only to Alice and k_b is known only to Bob. These integers act as the private DHE keys in the exchange. Alice then calculates her public DHE key:

$$P_a = g^{k_a} \bmod p.$$

Bob does a similar process to calculate his public DHE key:

$$P_b = g^{k_b} \bmod p.$$

P_a is sent to Bob and P_b is sent to Alice. Now that both parties have the other's public DHE key, they can compute the Diffie-Hellman shared secret, s. Alice computes s:

$$\begin{aligned} s &= (P_b)^{k_a} \\ &= (g^{k_b})^{k_a} \bmod p \\ &= g^{k_a k_b} \bmod p. \end{aligned}$$

Bob computes s independently of Alice:

$$\begin{aligned} s &= (P_a)^{k_b} \\ &= (g^{k_a})^{k_b} \bmod p \\ &= g^{k_a k_b} \bmod p. \end{aligned}$$

Both parties end up with the same shared secret. This result is possible due to the commutative property discussed in § 2.4.2. Although Eve is able to see p, g, P_a, and P_b, she cannot calculate s if the DLP is hard. The Diffie-Hellman shared secret becomes the premaster secret used in the TLS key generation (see § 2.2.3).

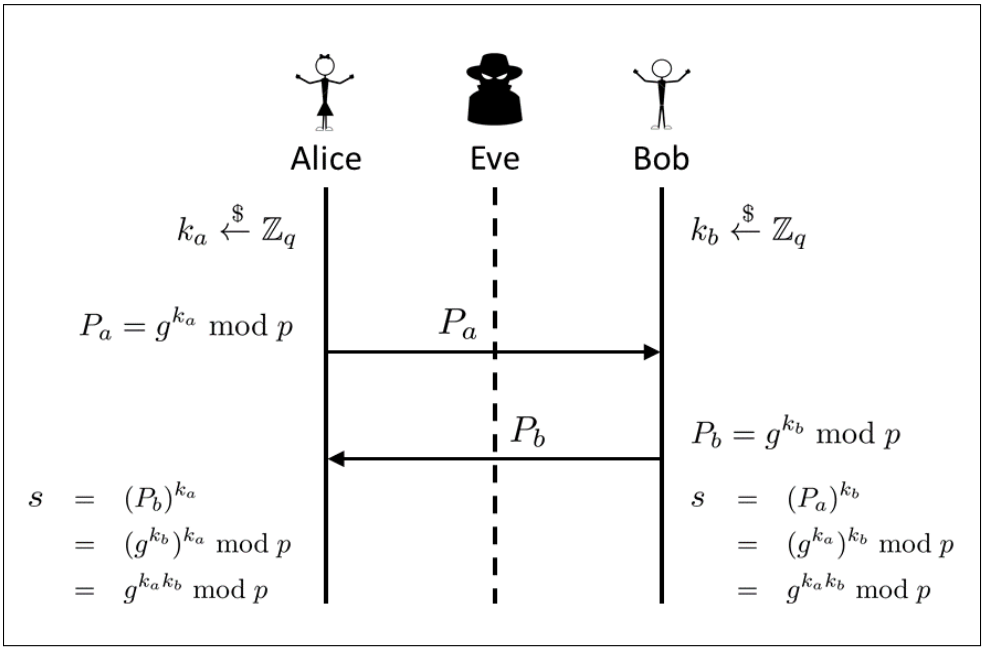

Alice
Eve
Bob
$k_a \xleftarrow{\$} \mathbb{Z}_q$
$k_b \xleftarrow{\$} \mathbb{Z}_q$
$P_a = g^{k_a} \bmod p$
P_a
P_b
$P_b = g^{k_b} \bmod p$
$s = (P_b)^{k_a} = (g^{k_b})^{k_a} \bmod p = g^{k_a k_b} \bmod p$
$s = (P_a)^{k_b} = (g^{k_a})^{k_b} \bmod p = g^{k_a k_b} \bmod p$

Figure 2.3: **Diffie-Hellman Key Exchange.** The finite field Diffie-Hellman key exchange involves two parties exchanging public keys and computing a shared secret.

The security of Diffie-Hellman is more specifically defined from the computational Diffie-Hellman (CDH) assumption and the decisional Diffie-Hellman (DDH) assumption. Properly chosen Diffie-Hellman domain parameters should satisfy these assumptions, ensuring the key exchange is theoretically secure.

Definition. (*Computational Diffie-Hellman (CDH) assumption.*) The CDH assumption states that given a random set of Diffie-Hellman domain parameters $\langle p,q,g\rangle$ forming $\mathbb{G}_q$, and elements $\langle g^a,g^b\rangle$ in $\mathbb{G}_q$, it is computationally intractable to find g^{ab}.

Definition. (*Decisional Diffie-Hellman (DDH) assumption.*) The DDH assumption states that given given a random set of Diffie-Hellman domain parameters $\langle p,q,g\rangle$ forming $\mathbb{G}_q$, and elements $\langle g^a,g^b,g^c\rangle$ in $\mathbb{G}_q$, it is computationally intractable to recognize a difference between g^{ab} and g^c.

Backdoors. A backdoor is a way to bypass security mechanisms, such as encryption and authentication from a cryptosystem. Although it can refer to a bypass installed for legitimate reasons such as troubleshooting, it is more often secretly exploited or installed by malicious designer Heidi. For the purposes of this work, a backdoor refers to a *maliciously installed* backdoor. We discuss creating, finding, and installing backdoors in Diffie-Hellman in Chapters 3 and 4.

2.4.4 Related Work

Inadequate DHE Parameter Validation. As mentioned in § 2.4.2, the discrete logarithm problem is hard for subgroups that are sufficiently large and not smooth. Not following these guidelines results in insecure implementations, which has been known for decades [59, 11, 82]. Despite this, many popular discrete logarithm implementations do little or no parameter validation, which will be discussed further in § 3.2. Valenta et al. [81] published work independently but concurrently to our paper [35], and it contained many similar results about the weak Diffie-Hellman parameters used in HTTPS and other protocols. Whereas our paper focuses on the possibility of backdoors stemming from small subgroups of hidden order, their work focuses on how the lack of parameter checking can be exploited in the context of Digital Signature Algorithm (DSA) style groups.

Other recent work exploiting poor parameter validation includes Bhargavan et al.'s 2014 [21] and 2015 [22] papers. The first paper demonstrated a triple handshake attack against TLS, which succeeded because the client did not check if the group order was prime. The second paper demonstrated a small subgroup attack against TLS and SSH, which succeeded because the public key was not validated and thus could be chosen in a deliberately small subgroup. Mavrogiannopoulos et al. [62] defined a TLS attack used when a server supports explicit Elliptic Curve Diffie-Hellman curves, which succeeded since the client can view the Elliptic Curve

Diffie-Hellman parameters as Diffie-Hellman parameters. Despite recovering the premaster secret, this attack is very limited as explicit Elliptic Curve Diffie-Hellman curves are not supported in the majority of TLS implementations due to their open-source nature.

Backdoors Based on Subgroups of Smooth Order. Our work with Diffie-Hellman discusses the possible existence of backdoor discrete logarithm groups (see § 4.2 and § 4.3). Henry and Goldberg [49] solved the discrete logarithm in some smooth order groups using a parallelized implementation of the Pollard's rho algorithm [70], and concluded that their implementation could be used to create a backdoor DL group.

In addition to Valenta et al. [81], Wong [85] recently published concurrent but independent work to us. Wong found composite DHE moduli over HTTPS in the wild, but our work reports on considerably more moduli across a wider range of protocols. In addition, the exploitation by Wong required both the client and server to prefer a DHE cipher suite, which limits the attack potential since current telemetry data [65] indicates DHE key exchanges account for at most 1% of TLS handshakes. In § 4.4.1 we describe how an attacker can exploit backdoored parameters to force a DHE cipher suite to be selected if both parties support it. Additionally we explain how one of Wong's backdoor constructions could be reversed in less operations than he expected. We also conducted a number of vulnerability disclosures and discuss vendor responses in § 4.6.

Backdoors Based on Number Field Sieves. In addition to work on backdoors based on smooth order subgroups, there has also been work on backdoors based on number field sieves. Lenstra [56] and Gordon [46] observed that even if it was established that a particular group had a sufficiently large prime order and that all relevant values were members of the group, it is not necessarily sufficient to ensure the hardness of the discrete logarithm problem if p was maliciously chosen to be "nice" in the context of the generalized number field sieve. Here, a backdoored prime modulus could be constructed using a polynomial of low-degree and constrained coefficients for the purposes of greatly accelerating the sieving and descent steps of a generalized number field sieve (GNFS). Given only p, a verifier would need to deduce this polynomial in order to establish the existence of a backdoor. This approach to building backdoored Diffie-Hellman parameters was previously considered too computationally intensive to perform in practice.

However, Fried et al. [42] recently demonstrated the creation of a 1024-bit backdoored prime modulus using the special number field sieve. Number field sieving can even be applied in some situations where the group was not attacker controlled. Adrian et al. [8] demonstrated a modified version of the GNFS, which they named Logjam, in which an attacker could recover private DHE keys from export strength (512-bit) groups.

2.5 X.509 v3 Certificates

As mentioned in § 2.2.3, X.509 v3 certificates are used for server authentication in TLS. This section discusses the need for certificates, the trust hierarchy of certificate chains, certificate generation and issuance, fields and extensions in a certificate, and the related work in this area.

2.5.1 The Need for Certificates

MITM Attacks on Diffie-Hellman. In § 2.4.3, we outlined how Diffie-Hellman key exchange could be used to securely compute a shared secret between parties Alice and Bob. It was assumed in that section that an eavesdropper, Eve, could not calculate the shared secret if the DLP was hard. However, that scenario does not stop Eve from establishing a Diffie-Hellman key exchange with both Alice and Bob.

First, Eve would intercept Alice's attempt to set up a key exchange with Bob, and complete a Diffie-Hellman key exchange with Alice such that their shared secret is s_{ae}. Eve would then initiate a Diffie-Hellman key exchange with Bob such that their shared secret is s_{be}. Since a shared secret is the basis for secure communication, Eve can now intercept Alice's messages to Bob, undo the security with s_{ae}, then redo the security with s_{be} before forwarding the messages to Bob. The same idea applies with Bob's messages to Alice. To prevent this MITM attack from happening, DHE parameters need to be signed as mentioned in § 2.2.3.

Digital Signatures. In § 2.2.3, we explained that DHE parameters are signed with the server's private key (corresponding to the public key on the server's certificate) before they are sent to the client. A digital signature scheme consists of three parts:

1. **Key Generation.** Key generation involves randomly generating a key pair (private signing key and associated public verification key). In TLS, the server is the one to generate a key pair.

2. **Signing Algorithm.** The message to be signed is hashed. The hash value is given to the signature algorithm along with the private signing key to produce a digital signature. In TLS with Diffie-Hellman, the DHE parameters with signature are sent to the client.

3. **Signature Verification.** The party who receives the signature verifies it with the public verification key. In TLS, the client verifies the server's signature on the DHE parameters, confirming the authenticity of the parameters.

Common signature schemes used in TLS are RSA and the Elliptic Curve Digital Signature Algorithm (ECDSA). While digital signature schemes prevents the MITM attack on Diffie-Hellman described previously, there is still a problem of confirming that the public verification key is from the server. Public-key certificates were created to solve this problem.

Certifying Public Verification Keys. A public-key certificate, or simply certificate, is used to certify the ownership of a public key such as a public verification key. In general, a X.509 v3 certificate [28] consists of a public key and identifying information about the owner of the public key, and it is signed by a trusted third party who issued the certificate [79]. In TLS, the client uses a certificate to attest to the authenticity of the public verification key. This attestation confirms that the key comes from the server, which means the signature and therefore the DHE parameters are also from the server. In the next sections, we discuss X.509 v3 certificates in more detail.

2.5.2 Chain of Trust

Public-Key Infrastructure. In the context of the Internet, X.509 certificates form the basis for a public-key infrastructure (PKI) to securely and efficiently certify public key owners as explained in § 2.5.1. This PKI establishes a trust hierarchy between a trusted third party (a certification/certificate authority, or CA), and an end entity who needs the certificate.

Chain of Trust. In the TLS Handshake, server certificates are typically arranged in a hierarchical chain. This chain of trust is demonstrated in Figure 2.4 using Facebook[2] and Twitter[3] as examples. The italicized names are the common names of each certificate (discussed in § 2.5.3). The trust chain contains three types of certificates:

A. **Root Certificate.** A certificate chain starts with a certificate from a trusted root CA [17]; clients come already installed with a list of root CAs to trust by default. Certificates from root CAs are aptly called root certificates. The specific list of root certificates trusted depends on the browser and operating system. For example, using Chrome on Windows employs the Microsoft root certificate store (shipped with Windows), but using Firefox on Windows employs the Mozilla store (shipped with Firefox). The root certificate store is also called the trust store. Root certificates are self-signed, meaning the certificate is issued by the same authority as its subject. This practice is only acceptable with root

[2] `https://www.facebook.com/`

[3] `https://twitter.com/`

certificates because they exist at the top of the trust chain and so cannot be signed by another authority [67].

Definition. (*Self-signed certificate.*) A self-signed certificate is one where the certificate's issuer is also its subject.

B. **Intermediate Certificate.** The second part of the certificate chain is one or more intermediate certificates. An intermediate certificate, from an intermediate or subordinate CA, is trusted because its issuer is a root CA [67]. The purpose of an intermediate certificate is to decrease the possibility of root certificate compromise by providing another layer of protection. There may be multiple intermediate certificates in a chain, but the last one is responsible for issuing a certificate to the end entity the requires it.

C. **Leaf Certificate.** The final certificate in the chain is the leaf, or end-entity, certificate. This certificate is the one issued to the end user or system, which for our purposes is a domain owner. Leaf certificates are discussed further in § 2.5.3.

2.5.3 Certificate Fields and Extensions

This section focuses on the fields and extensions present in X.509 v3 certificates, specifically leaf certificates. An example certificate from `google.com`, seen in Figure 2.5, is used to illustrate common fields and extensions.

Certificate Fields. X.509 certificate fields outline the basic structure of the certificate. We restrict our discussion to relevant fields.

(1) **Version.** See line 3 of Figure 2.5. The version field indicates which version of X.509 is used. At the time of writing, this is normally version 3.

(2) **Issuer.** See line 7 of Figure 2.5. The issuer field contains information about the certificate's issuer, collectively called the distinguished name (DN) of the issuer [76]. A DN is made up of attributes; common attributes include country, organization, and common name. Common names are explained in the subject field.

(3) **Validity.** See line 8 of Figure 2.5. The certificate is valid between the start and end dates specified in the validity field.

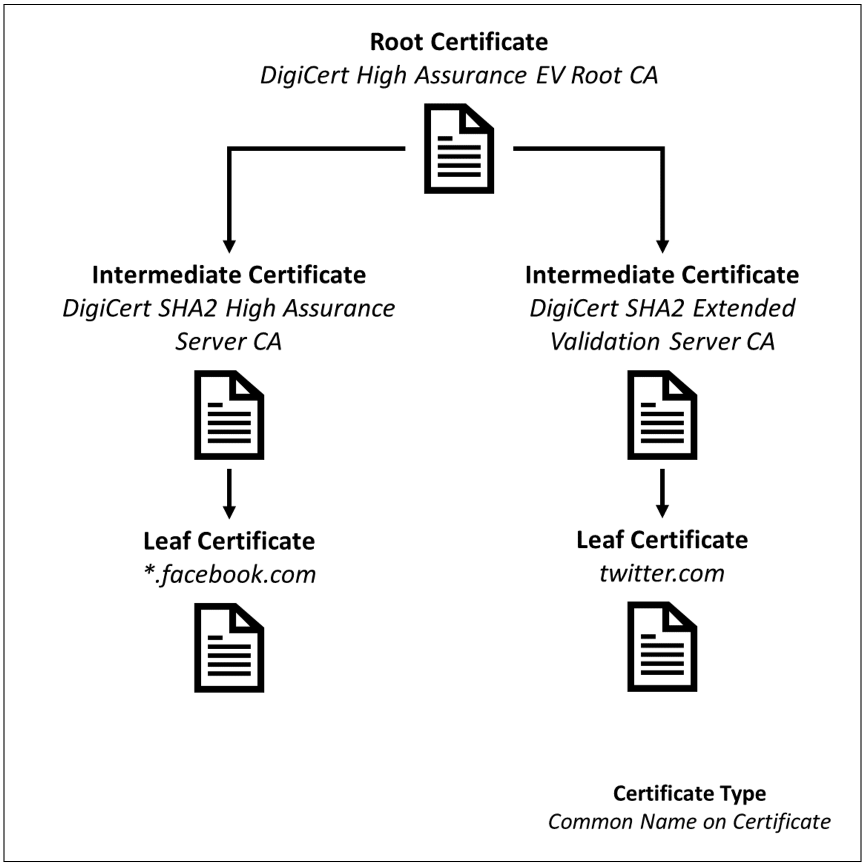

Root Certificate
DigiCert High Assurance EV Root CA
Intermediate Certificate
DigiCert SHA2 High Assurance Server CA
Intermediate Certificate
DigiCert SHA2 Extended Validation Server CA
Leaf Certificate
*.facebook.com
Leaf Certificate
twitter.com
Certificate Type
Common Name on Certificate

Figure 2.4: **Certificate Trust Hierarchy.** A chain of trust using Facebook and Twitter as examples.

(4) **Subject.** See line 11 of Figure 2.5. The subject field contains the DN of the certificate's subject, which is the entity that has the private key that pairs with the public key on the certificate [28]. The subject's DN contains the same possible attributes as the issuer's DN.

A relevant attribute for our research is the common name (CN). The common name attribute is a Fully Qualified Domain Name (FQDN), which is a complete domain name (i.e. specifies an exact host on the Internet, see § 5.2.1). It can contain a wildcard (see § 5.2.1), meaning FQDN contains an asterisk (*) at the far left [71]. As an example, the common name `*.google.com` covers `www.google.com` but does not cover `test.www.google.com`. Wildcards are implemented to allow a domain owner to require less certificates, but can be confusing for a user connecting to a domain that is not exactly specified by the certificate.

Definition. (*Common name.*) A common name (CN) is an attribute of a certificate's subject. This attribute is usually a Fully Qualified Domain Name (FQDN) or a domain with wildcard

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            6b:0c:76:d7:7a:a0:ae:e0
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2
        Validity
            Not Before: Jul 19 11:30:28 2017 GMT
            Not After : Oct 11 11:30:00 2017 GMT
        Subject: C=US, ST=California, L=Mountain View, O=Google Inc, CN=∗.google.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                PublicKey: (2048 bit)
                Modulus:
                    00:bc:6a:a7:b9:61:36:71:2e:1d:5d:79:4c:7a ...
                    (additional bytes omitted)
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Subject Alternative Name:
                DNS:∗.google.com, DNS:∗.android.com, ... (additional names omitted) ...,
                    DNS:youtube.com, DNS:youtubeeducation.com, DNS:yt.be
            Authority Information Access:
                CA Issuers  URI:http://pki.google.com/GIAG2.crt
                OCSP  URI:http://clients1.google.com/ocsp

            X509v3 Subject Key Identifier:
                F1:6A:43:32:4C:17:53:37:A9:01:44:40:85:DF:EA:78:ED:84:74:CB
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                keyid:4A:DD:06:16:1B:BC:F6:68:B5:76:F5:81:B6:BB:62:1A:BA:5A:81:2F

            X509v3 Certificate Policies:
                Policy: 1.3.6.1.4.1.11129.2.5.1
                Policy: 2.23.140.1.2.2

            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://pki.google.com/GIAG2.crl

    Signature Algorithm: sha256WithRSAEncryption
         04:3f:93:00:57:f0:c1:e5:0f:5e:f2:7f:fa:91:d0:30:62:f0 ...
         (additional bytes omitted)
```

Figure 2.5: **X.509 Certificate.** An example X.509 certificate of `google.com` obtained using OpenSSL's `s_client`.

such as *.example.com.

(5) **Subject Public Key Info.** See line 12 of Figure 2.5. The public key field contains the public key of the subject and its associated algorithm, along with domain parameters if needed.

Certificate Extensions. X.509 certificate extensions are possible in version 3 certificates, and were created to add more options to the basic structure. We restrict our discussion to relevant extensions.

(1) **Subject Alternative Name.** See line 22 of Figure 2.5. When a certificate needs to be valid for more domains than the common name specifies, the subject alternative name (SAN) extension is used. In the example above, the certificate for google.com covers additional domains such as youtube.com.

Definition. (*Subject alternative name.*) The subject alternative name (SAN) extension is used in a certificate if it needs to cover more domains than the one(s) specified in the CN.

Server Name Indication. Server Name Indication (SNI) was created since name-based virtual hosting allowed for multiple websites at one Internet Protocol (IP) address, but makes use of the HTTP header that happens after the TLS handshake. In the case where multiple HTTPS websites are hosted on a server using one IP address, they must all use one certificate unless SNI is used. SNI is an extension to TLS [39] that is frequently used by browsers to specify the website to connect with before the TLS handshake, which allows a web server to have multiple certificates on one IP address. SNI is now almost universally adopted; a recent study by Content Delivery Network (CDN) provider Akamai[4] showed that 99% of HTTPS requests over Akamai's network are done by clients supporting SNI [66].

Certificate Errors. If a user connects to a website and the browser detects a problem with the certificate, the browser will display a certificate error. Generally the user has the option to ignore the error and continue to the website, and recent studies show that between 33% and 56% of users do ignore the warning [10, 41]. If the website uses HSTS, users are not permitted to click through the warning as explained in § 2.2.4.

We focus the discussion on two certificate errors which make the certificate invalid: self-signed certificates and name mismatch errors. Self-signed certificates are needed for root certificates (see § 2.5.2), but are not considered valid for leaf certificates. A name mismatch error

[4] https://www.akamai.com/

occurs when the user accesses a website over HTTPS and the website is not covered by the certificate. More specifically, the website must be present in the CN and/or SAN of the certificate, either as an exact match or as a wildcard match.

Definition. (*Name mismatch error.*) A name mismatch error is one type of error that invalidates a certificate. This error occurs when none of the names in the CN and SAN of the accessed website's certificate cover that website through an exact or wildcard match.

We now discuss the related work in the area of name mismatch errors.

2.5.4 Related Work

X.509 Certificates from Internet-Wide Scans. Holz et al. [53] conducted various active scans, including of the IPv4 (IP version 4) space, along with passive scans of a research network to investigate X.509 certificates in HTTPS. One of their findings was that around 80% of the investigated certificates had name mismatch errors, but did not investigate further beyond a few unusual names in the CN and self-signed certificates. A similar study by Eckersley and Burns [40] had been done the year prior, albeit on a slightly smaller scale. Taking all domains from sets such as .com, .net, .org domains, Ristić [75] scanned 119 million domains to investigate their certificate configurations. However, to narrow the investigation, he did not examine certificates with name mismatch errors.

Over approximately a year, Durumeric et al. [37] conducted 110 scans of the IPv4 space to study the behaviour of CAs, and as a side effect discovered some unusual names in the CN and SAN similar to Holz et al. [53]. More recently, VanderSloot et al. [83] used multiple measurement techniques to create the most comprehensive HTTPS certificate list possible. Their data sets included IPv4 scans and all domains from .com, .net, and .org. They determined that IPv4 scanning alone misses approximately 65% of websites because many sites require SNI.

TLS Configurations from Internet-Wide Scans. Holz et al. [52] conducted active scans of IPv4 space and passive scans of a university network to investigate the TLS and STARTTLS configurations of mail and chat protocols such as SMTP and XMPP. Although certificate chain validity was investigated in detail, name matching could not be studied as they scanned IP addresses instead of domain names. Akhawe et al. [9] used passive scanning to investigate common TLS warnings and provide suggestions to decrease the prevalence of these errors. They found that about 20% of certificate errors were name mismatch errors. They further categorized name mismatch errors into groups for the purpose of suggesting improvements for browsers.

TLS Configurations from Specific Website Groups. Some recent related work has also focused on narrowed website groups within HTTPS. SSL Pulse[5], a follow-up project to Ristić's 2010 survey [75], outlines TLS implementation issues for approximately 150 000 popular sites every month. It does not examine name mismatch errors. Kranch et al. [55] found basic errors in many sites' HSTS implementations, surveying sites on HSTS preload lists and in the Alexa[6] Top Million list. Liang et al. [58] investigated sites from the Alexa Top Million list that had ties to one of 20 CDNs, and found many issues with HTTPS implementation by CDNs.

[5] `https://www.ssllabs.com/ssl-pulse/`

[6] `http://www.alexa.com/`

Chapter 3

Diffie-Hellman Backdoors: Mathematical Construction

A version of this chapter has been published as part of [35].

3.1 Overview

In § 2.4.3, we explained the Diffie-Hellman key exchange is a discrete logarithm implementation, with its security depending on the selection of Diffie-Hellman parameters, $\langle p,q,g \rangle$. In § 2.4.2, we clarified that the discrete logarithm problem (DLP) is computationally hard when the order q is sufficiently large and not smooth. Validating Diffie-Hellman parameters is necessary to ensure DLP hardness; if the DLP is efficient, the Diffie-Hellman key exchange is insecure which undermines the entire security of the TLS connection.

In this chapter, we found that many discrete logarithm implementations perform little or no validation on Diffie-Hellman parameters. We demonstrate this lack of validation by successfully connecting to implementations using DL, such as Chrome, with optimally weak parameters (i.e. parameters for which the DLP is efficient). We then investigated weak parameters further in the context of *backdoors*: an attacker could construct backdoored Diffie-Hellman parameters so that the DLP is both efficient and *appears* to be inefficient. We outline a backdoor construction that would accomplish these goals and contrast it with Wong's [85] concurrent but independent backdoor proposal.

This chapter contains three sections: the parameter hygiene of discrete logarithm implementations, including poor validation techniques and unnecessary information leaking, is discussed in § 3.2; demonstrations showing poor validation in practice is discussed in § 3.3; and backdoor constructions are discussed in § 3.4.

3.2 Parameter Hygiene in DL Implementations

In this section, we discuss the poor parameter hygiene found in discrete logarithm (DL) implementations, including a lack of validation checking and a tendency to work in a group that breaks the DDH assumption defined in § 2.4.3.

3.2.1 Missing Validation Checks

Verifying the validity of the domain parameters is sufficient to detect the kinds of weakened or backdoored parameters considered by this thesis. However, most of the software implementations we examined skip one or more validity checks:

- **Length**: Check that $|p|$ and $|q|$ are sufficiently large (i.e. $|p| \geq 2048$ bits, $|q| \geq 224$ bits as per current NIST guidelines [14]);

- **Primality**: Check p and q are both prime;[1]

- **Group Order**: Check $q|(p-1)$. No mechanism is provided in TLS to communicate group order [32, 72];

- **Group Membership**: Check any asserted group element (i.e. generator g, public key, etc.) is an element a of the group $\mathbb{G}_q$. Specifically, check $1 < a < p-1$ and $a^q \bmod p = 1$. Note $a = p-1$ is explicitly excluded by the associated NIST standard [13], since it always only has an order of 2, regardless of the choice of p. Safe prime groups working in $\mathbb{Z}_{p_s}^*$ can omit the exponentiation by the group size, since all elements $1 < a < p_s - 1$ are part of this group.

Most finite-field based DL implementations we examined inherently treat domain parameters as trusted. Many of the necessary checks (e.g. primality, group membership, etc.) are done when the parameters are generated, but at no point thereafter. As an example, recall the digital signature scheme outlined in § 2.5.1 – the OpenSSL implementation of the Digital Signature Algorithm (DSA) does not check parameters during key generation, signing, or verification and we were able to construct accepted universal forgeries with maliciously constructed parameters. This would not pose a problem in most cases since usually the signer is expected to generate their own parameters, but this strategy does not always work out.

One related example arose in OpenSSL when using non-safe prime groups (i.e. X9.42 groups [1]) in Diffie-Hellman key exchanges, where the server's private Diffie-Hellman key was reused (in fixed/static Diffie-Hellman modes) or when exponents were reused across more

[1] Technically q only must contain a sufficiently large prime factor.

than one connection for efficiency. By not checking the received client public Diffie-Hellman key was in the intended group (i.e. in $\mathbb{G}_q$), a malicious client could partially or fully recover the server's private Diffie-Hellman key. This resulted in CVE-2016-0701 [2]. Now OpenSSL performs a group membership test of client public Diffie-Hellman keys on the server side, but only when an X9.42 group is ostensibly in use. In the case of maliciously injected parameters, OpenSSL will still successfully proceed with Diffie-Hellman key agreements using composite moduli, small groups, and other weak parameters.

3.2.2 Working in $\mathbb{Z}_p^*$ with Generator of Order $2q$

Many of the finite-field discrete logarithm implementations we examined work in $\mathbb{Z}_p^*$, as opposed to a prime order subgroup. The trend seems to have begun with the Handbook of Applied Cryptography (see Section 4.6.1 of [63]), and many implementations explicitly cite it. For example, OpenSSL generates Diffie-Hellman parameters that intentionally work in $\mathbb{Z}_p^*$, noting in a code comment that their generator of $\mathbb{Z}_p^*$ "will generate either an order-q or an order-2q group, which both is OK."[2] However, the comment further goes on to say "[it's] just as OK (and in some sense better) to use a generator of the order-q subgroup." One reason that working in $\mathbb{G}_q$ is better than working in $\mathbb{Z}_p^*$ is that with a generator of order $2q$, the latter needlessly leaks a bit of the private DHE key since the discrete logarithm of 2-order subgroup is easily computed. This generator selection breaks the DDH assumption since it can now be distinguished if the private DHE key is even or odd.

Officially, there is little risk to the CDH assumption (see § 2.4.3) if $p-1$ contains a sufficiently large factor and full length exponents are used. In this case, the private exponent is also sampled from $\mathbb{Z}_p^*$, although Boneh et al. suggest related attacks in this setting [24]. A major risk comes about when developers use short exponents (e.g. 160, 224, or 256 bits) in the interest of performance, and the Pohlig-Hellman attack [68] may become applicable depending on the subgroup structure.

But we argue working in $\mathbb{Z}_p^*$ with a generator of order $2q$ is simply bad parameter hygiene; there is no reason to leak even one bit of information. In addition, it sets a bad precedent for developers who might be tempted to apply this thinking to seemingly similar but subtly different situations. For example, we found the libgcrypt,[3] pycrypto,[4] and bouncycastle[5] implementations of ElGamal all by default work in $\mathbb{Z}_p^*$ with a generator of order $2q$. This generator selection is conspicuous since it breaks the DDH assumption and hence semantic security as

[2] https://github.com/openssl/openssl/blob/master/crypto/dh/dh_gen.c

[3] https://gnupg.org/software/libgcrypt/index.html

[4] https://pypi.python.org/pypi/pycrypto

[5] https://www.bouncycastle.org/

explained earlier.

GNU Privacy Guard (GPG), for example, uses libgcrypt and the authors confirmed their GPG public ElGamal encryption keys all leak one bit of their respective private keys. Although this does not lead directly to an attack because the plaintext in this setting is (largely) a random value, it is both unnecessary and potentially a sign of additional cryptography issues. For example, GPG makes curious parameter choices and an ElGamal key pair at the 2048-bit level consists of a prime in which $p-1$ consists of a 340-bit private key in a 235-bit subgroup. Although many of the applications using these libraries seem not to require DDH, focusing instead on things such as encrypting random nonces, neither do the libraries come with the warning that the implementations are not semantically secure as one might nominally expect of an ElGamal implementation. This is probably acceptable when encrypting a session key, but is not as acceptable if the library were to be used as part of an implementation of a cryptographic voting system encrypting ballot choices. For example, Chang-Fong and Essex recently exploited small subgroups in Helios [27], an Internet voting system that provides end-to-end cryptographic verification. Finally we note the use of $\mathbb{Z}_p^*$ with a generator of order $2q$ is not universal. In contrast to the more ad hoc approach to parameter generation of many implementations, standardized parameters such as the Modular Exponential (MODP) [57] and Oakley [48, 54] safe prime groups use generators that do not leak a bit. We consider working with safe prime groups with short exponents to be a good balance between security and efficiency.

3.3 Successful Connections with Weak Parameters

In this section, we demonstrate the lack of parameter validation discussed in § 3.2 by successfully serving weak parameters to DL implementations.

3.3.1 Connections with OpenSSH

A backdoored modulus may possibly remain undetected for longer if the weak modulus at least looks valid, e.g., does not end with an even digit. To demonstrate this, we investigated the visual similarity between a safe prime modulus and a deliberately weak modulus, and showed that lack of proper validation allows software implementations to connect with both moduli. As a demonstration, we modified the OpenSSH `\etc\moduli` file to use a deliberately weak modulus. The default OpenSSH moduli file consists of safe primes with short generators such as 2 or 5. Although the software does not check group validity, an attack in the context of a version update should allow the parameters to pass casual inspection. The attacker – in this

```
# Time Type Tests Tries Size Generator Modulus

20160522030737 2 6 100 2047 2 DB36277B45EA5615C782C08BF6A290A3D61E6B9690E4A147042113FC1BFC0AE
EC5FB0FF82FC1FEA86E273F667EC387FEF3421FFFC617A70C34B1987986C6B35C715713914AB75932A3D1942ECC0F
324D81BF00D59916B3BFDC7BA432AF5C5DFCF30BF4A2C80B8CA52A9B80E989D3A852BD81A8BD3ADC97497F43C6F0A
90882D9CFA165CF1F735C96428BF9BC32A58B71CF1D4FD48A6D2C616E91BB6E07C5CB0DF0C59DAF79D659C6E53007
843497BBEE5B341D27DE2E2543B8DFEB4DDAE6328EAD441C3F36509C1FA689FE494B0426ADCAF9E567A1C5A330168
9C5CCC55EC4002FAA5D254C2F3C0F8636BEA7019D1CD212B74EE4F273E0B9997720E8AEC5D76B

20160522030739 2 6 100 2047 2 8A4F17035FD10C065879FCC6C6632C15F18E15B6F88CAE2BA8C40D23E3DC2FD
68E8897E12F9FD6C3447B72C1595B2EF56C103162BB6C15AA64761C4258E56D47FE156832F6BB4273A106D2E6310A
9D5E54C497517A928A988A359FB0032BED2FEF690487F6AC6F0B3659A43643A316F601DE73E563F7BC2C37A67E751
DE1916B08FBE92FB9E32E35DC5FD051E9EBC4B2256BC4021DACD2CA816F46C7A5C5D1B298A259C925AB0DC404BCF7
2FDAF04C849DCA4C2F6576FCC586A5B942188312787D971D9BE6D70896A8E8458F3D75D6C8F97CE289688A175F699
B938DBFFC7A349D4130558794936E67C349EF96B83517CB647BADBF012E9BF1B4890E72B70849
```

Table 3.1: **OpenSSH `moduli` file**. One modulus is a valid safe prime (ostensibly) generated by developers. The other is a smooth composite allowing efficient discrete logarithms. OpenSSH will successfully connect with either.

case malicious designer Heidi – wants to create parameters that also have short generators (and thus are valid *looking*), but are still efficient to solve. Non-safe prime groups are unlikely to have short generators of small subgroups, and large generators (i.e. the same length as the modulus) would be overtly suspicious. Since OpenSSH does not verify the primality of the modulus, Heidi can instead work with smooth composite moduli. Here discrete logarithms can be made to be efficiently solvable for any generator of any subgroup.

As an example, we set p as the product of all primes up to 1471, excluding 2 and 5 (so it is not obviously prime from inspection in base 2 or 10). This number is 2043 bits and has 231 factors. Multiplying it by 19 will bring the length to a standard 2048 bits. In this case, one of the factors will be (19^2). Table 3.1 shows an example of a safe prime modulus and our smooth composite modulus. The lack of proper validation described in § 3.2.1 means OpenSSH connects with both the safe prime modulus and our composite modulus designed to allow efficient DLs. The discrete logarithm of a number relative to an arbitrary base (e.g. 2) can be computed individually across each of the factors of p and reassembled using the Chinese remainder theorem (CRT). The discrete log in each of the subgroups can be pre-computed. Computing a discrete log, therefore, can be reduced to 231 look-ups in this dictionary, followed by a single CRT of 231 congruences. Implementing this in Sage[6] we were able to compute discrete logarithms in 4 ms on a laptop.

3.3.2 Connections with Browsers

We determined ephemeral finite field Diffie-Hellman (DHE) support by browsers, then tested their parameter validation by serving them weak DHE parameters. Many major web clients still

[6] `http://www.sagemath.org/`

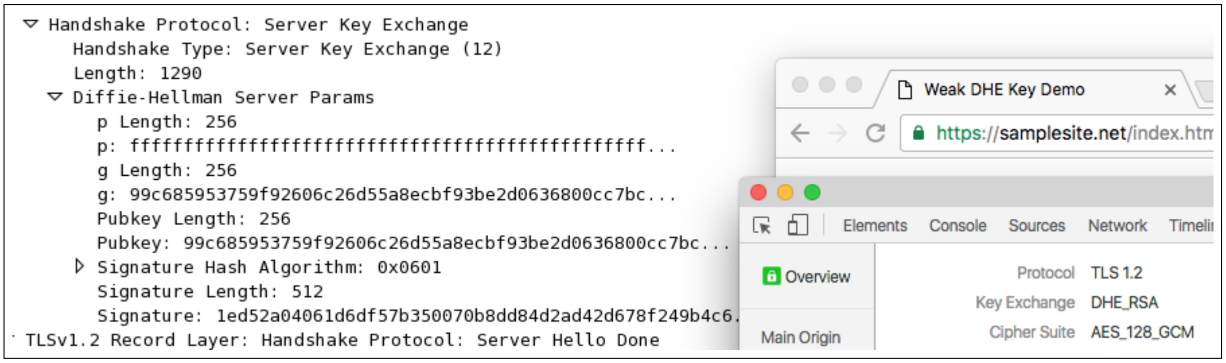
support DHE, although Safari and Chrome have removed DHE support. At the time of writing, Chrome was still in the process of removing support [18], but in the interest of interoperability connected with DHE if it is the only key exchange mode offered by the server. First it sent the `ClientHello` without DHE cipher suites, and if that fails it attempted again with DHE cipher suites added back in. This was largely in response to the difficulty in guaranteeing large moduli bit lengths following the results of Logjam [8], which we discuss further in § 2.4.4. Additional factors include the slower performance relative to ECDHE, although this gap is exacerbated by the predominance of safe prime implementations using full-length exponents. At the time of writing, DHE was still supported in approximately 87% of browsers,[7] though this dropped steeply to about 22% after Chrome removed support. Based on our own survey approximately 26% of servers support DHE over HTTPS (see § 4.2.2 for more information).

We tested major web browsers to see to what extent they would accept weak DHE parameters. We configured OpenSSL's `s_server` to accept only DHE cipher suites and serve custom generated Diffie-Hellman parameters. We wrote a program to generate malicious DHE parameters and encode them in OpenSSL's ASN.1 / pem format. We tested a number of different composite moduli as well as non-safe prime groups of low order.

Tested browsers include Chrome, Safari, Firefox, Internet Explorer, and Microsoft Edge. At the time of testing all browsers still supported DHE cipher suites. In each of the browser cases, the connection was successfully established with weak parameters or composite moduli, and no warnings were shown except in certain special cases. For example, Chrome generated an error when served moduli below 512-bits, even prior to the Logjam [8] disclosure.

Interestingly browsers do perform a kind of limited primality test on the modulus and will reject even numbers. When presented with an even modulus, most browsers would generate an error, then switch to RSA for key exchange and proceed with the connection. In all cases the browsers would not accept obviously trivial values such as public DHE keys or generators equalling 1 or $p-1$, meaning they *do* defend against working in the trivial group $\mathbb{G}_2$. The next smallest possible subgroup is one of order 3, in which the server public DHE key can be either 1, g or g^2. Working in this group will generate a browser error approximately one third of the time (i.e. when $g = 1$), but in the interest of reliability many browsers would attempt the connection several more times and would succeed with high probability, and no errors would be displayed to the user. A 2-bit key is an extreme example, and a real designer Heidi can make failure extremely unlikely by selecting a slightly larger subgroup while still keeping discrete logarithms computable in real-time.

[7] `https://www.w3counter.com/trends`



```
▽ Handshake Protocol: Server Key Exchange
    Handshake Type: Server Key Exchange (12)
    Length: 1290
  ▽ Diffie-Hellman Server Params
      p Length: 256
      p: ffffffffffffffffffffffffffffffffffffffffffffffff...
      g Length: 256
      g: 99c685953759f92606c26d55a8ecbf93be2d0636800cc7bc...
      Pubkey Length: 256
      Pubkey: 99c685953759f92606c26d55a8ecbf93be2d0636800cc7bc...
    ▷ Signature Hash Algorithm: 0x0601
      Signature Length: 512
      Signature: 1ed52a04061d6df57b350070b8dd84d2ad42d678f249b4c6.
TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
```

Protocol TLS 1.2
Key Exchange DHE_RSA
Cipher Suite AES_128_GCM

Figure 3.1: **Two-bit Security in TLS**. A successful DHE connection in Chrome using a generator of order 3. During this run the generator happened to equal the public DHE key, indicating the private DHE key was congruent to 1 mod 3.

As a concrete example we used the following parameters in our browser test:

$$\begin{aligned} p &= 2^{2048} - 1557 \\ g_3 &= 2^{(p-1)/3} \bmod p \end{aligned}$$

Here p represents the largest 2048-bit prime and g_3 is a generator of a subgroup of order 3 (i.e. the smallest possible non-trivial subgroup a browser would need to perform validation). As an illustration in Figure 3.1 we show a successful connection in Chrome with the server presenting the parameters $(p, g_3, y = g_3)$. In the Developer Tools,[8] Chrome warns that DHE is deprecated, but does not notice the weak group. This result is expected, as TLS contains no explicit field for communicating a group's order.

In summary, the browsers we tested were unable to defend against a variety of weak parameters (small or smooth order), as well as backdoored groups involving composite moduli. The limited forms of checking that are performed are interesting from our perspective, as they constitute a kind of tacit acknowledgement that parameter validation is important – just so long as it is efficient.

3.4 Backdoor Construction

Working in small subgroups is efficient from the malicious designer Heidi's perspective, but comes with two downsides: (1) others can also exploit the weak group, and perhaps more importantly (2) strong evidence exists that the parameters are compromised. A more interesting scenario is to backdoor the modulus such that only Heidi can exploit it while making its very

[8] https://developer.chrome.com/devtools

existence a matter of speculation. In this setting Heidi can use a composite (e.g. RSA) modulus to construct a backdoor instance of the discrete logarithm problem. Let $n = pq$ for large primes p, q with the number of generators for the n-order group being $\phi = (p-1)(q-1)$. The idea is to work in small subgroups of hidden and smooth order, such that $(p-1)$ and $(q-1)$ contain smooth factors. A generator is then selected so as to have reasonably low order modulo p and q respectively, allowing the person knowing the factorization of n to solve several independent and efficient discrete logarithms.

3.4.1 Related Constructions

Concurrent and independent to us, Wong [85] also proposes using a hidden subgroup of a composite modulus in the context of backdoored Diffie-Hellman key agreement. Let $p = 2p_1p_2+1$ and $q = 2q_1q_2+1$ where p_1, q_1 are sized small enough to allow efficient computation of the discrete log in subgroups of order p_1 and q_1, but large enough to prevent brute forcing the discrete logarithm in a subgroup of hidden order p_1q_1, while p_2, q_2 are large so as to prevent factorization attacks, such as Pollard's p-1 attack [69]. Let the length of p_1 and q_1 be ℓ (i.e. $|p_1|= |q_1|= \ell$). A generator g is chosen of the unique subgroup $\mathbb{G} < \mathbb{Z}_n^*$ of order p_1q_1.

The order of g has length 2ℓ. The orders of g modulo p and q respectively are ℓ-bits in length each. Computing a discrete logarithm separately modulo p and q takes $2^{\frac{\ell}{2}}$ operations each using general discrete logarithm algorithms (e.g. Pollard's rho [70], etc.). With knowledge of the backdoor, therefore, the attacker can compute a discrete logarithm in $2^{\frac{\ell}{2}+1}$ operations. Without knowledge of the group order, Wong argues an attacker would require 2^{ℓ} operations to compute a discrete logarithm. As an example, Wong suggests that if g had an order of 200 bits in length (i.e., where $\ell = |p_1|= |q_1|= 100$), then an observer would require 2^{100} operations to compute a discrete logarithm, while an attacker could solve the discrete logarithms separately modulo p and q, requiring $2 \cdot 2^{\frac{100}{2}} = 2^{51}$ operations.

This expectation, as it turns out, is false as shown by Coron et al. [29] in the context of the cryptosystem due to Groth [47], which works in small RSA subgroups of hidden order. Groth's construction is effectively identical to Wong's backdoor discrete logarithm construction, except is being applied in the context of an encryption scheme. Once again let $n = pq$ for $p = 2p_1p_2 + 1$ and $q = 2q_1q_2 + 1$ for p, q, p_1, p_2, q_1, q_2 prime, and let g generate the unique subgroup $\mathbb{G} < \mathbb{Z}_n^*$ of order p_1q_1. Let h generate a subgroup of order $p_1p_2q_1q_2$. The values (n, g, h) form the public key. The values (p_1, q_1) form the private key. A message m is encrypted as follows:

$$\mathsf{Enc}(m) = g^r h^m \bmod n.$$

for random r. Decryption for a ciphertext, c, is accomplished as follows:

$$\mathsf{Dec}(c) = c^{p_1q_1} = (g^r)^{p_1q_1}(h^m)^{p_1q_1} = (h^{p_1q_1})^m \bmod n.$$

The discrete log of $(h^{p_1q_1})^m$ is computed to recover m. This can be efficient if m is small, although Groth also proposed a variant in which p_2 and q_2 are smooth, allowing for the discrete logarithm to be efficiently computed using Pohlig-Hellman [68]. The best attack proposed by Groth [47] factorizes n in time $\mathscr{O}(2^\ell)$, and works as follows. Recall g has order p_1q_1 and that $g^{p_1} \equiv 1 \bmod p$ and that $g^{q_1} \equiv 1 \bmod q$. For the greatest common divisor gcd, this gives

$$\gcd(g^{p_1} - 1, n) = q$$

and

$$\gcd(g^{q_1} - 1, n) = p.$$

Thus n can be factorized by computing $\gcd(g^i - 1, n)$ starting at $i = 2^\ell$ and incrementing until a factor is found, requiring at total of $\min(p_1, q_1) - 2^\ell$ operations. Note this approach is independent of the size of factors p_2 and q_2,

Similar to Wong, Groth proposed $\ell = |p_1| = |q_1| = 100$ as a trade-off between security and efficiency. Coron et al., however, demonstrated an attack on Groth's scheme recovering the factors of n in time $O(2^{\frac{\ell}{2}})$ instead of the expected $O(2^\ell)$. Notice here that g in Groth's scheme has the same order as g in Wong's scheme, and thus any attack on Groth's scheme that can recover the factors of n based on g can be directly applied to Wong's scheme revealing the backdoor. Coron et al. proposes Groth's scheme use $\ell \geq 160$. This is problematic if applied to our backdoor setting, since it would require the backdoor owner to compute two discrete logarithms on the order of 2^{80} operations.

3.4.2 Our Backdoor Construction

Similar to Groth's attack, Coron et al.'s attack exploits the overall order of g, but cannot directly exploit the order's factorization (since it is unknown). Our strategy, therefore, makes the overall order of g large enough to make factorization attacks infeasible, while smooth enough to still allow efficient computation of DLs by the backdoor owner.

Let $p = 2p_1 \dots p_k r_p + 1$ and $q = 2q_1 \dots q_k r_q + 1$ for prime p, q. Let each p_i, q_i be distinct, randomly chosen primes of bit length ℓ. Let r_p, r_q be distinct randomly chosen primes. We choose g to generate a group $\mathbb{G} < \mathbb{Z}_n^*$ of order $p_1 \dots p_k q_1 \dots q_k$, which gives g an overall order of $2k\ell$ bits.

We size ℓ to be large enough to preclude factorization of n using Pollard's $p-1$, while small enough that solving discrete logarithm instances in subgroups of order approximately 2^ℓ is efficiently computable. Using Pollard's $p-1$ factorization method, n can be factored as follows. Choose some $a \xleftarrow{\$} \mathbb{Z}_n^*$. Let ρ_i be the i-th prime. For each $\rho_i < 2^\ell$:

1. Set $a \leftarrow a^{\rho_i} \bmod n$

2. If $gcd(a-1,n) \neq 1$ and $\neq n$, output factor, otherwise continue.

Factorization is guaranteed after all primes $\rho_i < 2^\ell$ have been exponentiated in, corresponding to approximately $\mathsf{li}(2^\ell)$ modular exponentiations, where $\mathsf{li}(\cdot)$ is the logarithmic integral. Henry and Goldberg [49] studied solving discrete logarithms in smooth-order groups using optimized GPU implementations, and suggest $\ell = 55$ as sufficient, requiring 1500 years of (non-parallelizable) wall-clock time to factor n, while requiring less than two minutes to compute the discrete logarithm with knowledge of the backdoor.

We size k to be large enough to preclude factorization of n based on the order of g (as in Coron et al.'s attack), i.e., $2^{\frac{k\ell}{2}}$ operations is computationally infeasible. Following Coron et al.'s suggestion we have $k\ell \geq 160$. As a concrete parameter choice, let p,q each be 1024-bit primes where $p = 2p_1p_2p_3r_p + 1$ and $q = 2q_1q_2q_3r_q + 1$ where p_1, p_2, p_3, q_1, q_2 and q_3, are distinct, random 55-bit primes and r_p, r_q are distinct, random primes of a length sufficient for p,q respectively to be 1024 bits. A generator g is chosen of order $p_1p_2p_3q_1q_2q_3$. Given a public Diffie-Hellman key $g^x \bmod n$, recovering private key x requires 6 separate discrete logarithms to be computed in subgroups of order 2^{55}, for a total of approximately $6 \cdot 2^{\frac{55}{2}} \approx 2^{30}$ operations.

Plausible Deniability. One of the most desirable aspects of this attack paradigm is the ability for malicious designer Heidi to construct a discrete-log backdoor while maintaining plausible deniability. It is easy to tell that a modulus is composite (when you're looking), but determining group structure without knowledge of the factorization, and hence the likelihood of the existence of a backdoor, can be made to be computationally infeasible. As we explain in § 4.6.5, none of the vendors we contacted about the composite moduli we discovered were able or willing to either confirm or deny the existence of a backdoor – precisely as Heidi might hope!

One possible explanation for the origin of a composite modulus is that it was simply a random number chosen by accident, or perhaps began as a prime and had a digit or two flipped in an editor. In this case we would expect the resulting value to have a distribution of factors similar to that of a random composite number. We discussed setting $n = pq$ for large primes p,q, but this might arouse suspicion, beyond simply being composite, because it would contain no small factors. Small factors up to some bound b may be recoverable using elliptic curve factorization, and the probability that a random composite number is b-rough (i.e. contains no factors smaller than b) could be used as evidence toward the determination of the existence of a backdoor. One option would be for Heidi to use an RSA modulus as before but multiply in a sequence of naturally increasing factors up to bound b. We leave a heuristic for creating convincing random-looking but backdoored moduli for future work.

Chapter 4

Diffie-Hellman Backdoors: TLS and STARTTLS Presence

A version of this chapter has been published as part of [35].

4.1 Overview

In § 3.2, we outlined the lack of parameter validation by DL implementations, which fail to check basic properties such as moduli primality. We further demonstrated this lack of validation in implementations such as Chrome in § 3.3. Since browsers such as Chrome could accept weak DHE parameters, we outlined a mathematical construction for backdoored DHE parameters in § 3.4 that would allow an attacker to efficiently compute the DL of the parameters while keeping the backdoor deniable.

In this chapter, we investigated the possibility of backdoored DHE parameter use in TLS and STARTTLS. We conducted scans of the IPv4 space in both mail and web protocols to search for composite and non-safe prime DHE moduli, and found hundreds and millions of composite and non-safe prime moduli respectively. We additionally looked for such moduli in over 100 open-source projects. We factored some of the composite and non-safe prime moduli found and were able to recover a significant portion of the private DHE key in some cases. To increase the attack space, we proposed a MITM attack to force DHE in TLS 1.2 and below, and then discussed possible attack vectors for placing DHE parameters for use. Finally, we disclosed the composite moduli to companies and proposed mitigation strategies.

This chapter contains six sections: scans for composite DHE moduli are discussed in § 4.2; the other DHE testing, such as non-safe prime and open-source project investigations, are discussed in § 4.3; the MITM attack is discussed in § 4.4; attack vectors for placing DHE parameters are discussed in § 4.5; company disclosures are discussed in § 4.6; and mitigation strategies are discussed in § 4.7.

4.2 Composite DHE Moduli

This section outlines the composite DHE moduli found in protocols such as HTTPS.

4.2.1 Overview of Affected Protocols and Countries

Methodology. In order to find potential backdoors in discrete logarithm implementations, we collected Diffie-Hellman data from two sources. For HTTPS, we downloaded Censys[1] IPv4 scans [36] where only DHE cipher suites were offered by the client. Censys routinely collects this data using ZGrab[2] (an application-layer scanner) and ZMap[3] (a network scanner). For DHE-only scans in SMTP/S, POP3/S, and IMAP/S, we used ZGrab to run our own scans due to its fast performance. We investigated both non-safe and composite DHE moduli in HTTPS, and focused on composite moduli only in SMTP/S, POP3/S, and IMAP/S. This section focuses on composite moduli; non-safe prime moduli are discussed in § 4.3.1.

Affected Protocols. Overall, there were over 500 IP addresses in 31 countries using potentially backdoored composite moduli. A summary of moduli properties and the affected protocols are seen in Table 4.1. Out of the seven protocols investigated, composite moduli were found in five: HTTPS, IMAPS, POP3S, SMTP, and SMTPS. Almost all of the moduli were one of two numbers: a 512-bit modulus used in SMTP or a 2048-bit modulus used in HTTPS. This recycling of parameters is common practice; while it does not directly suggest backdoor use, having the same backdoor in hundreds of IP addresses is advantageous for an attacker. At the very least, this moduli reuse proves that weak DHE parameters are used in the wild due to lack of Diffie-Hellman parameter validation. Table 4.1 also shows three moduli with non-standard lengths of 4255-, 1102-, and 904-bits, indicating further carelessness in parameter choice.

Affected Countries. To see the impact of these composite moduli, we determined each IP address' location using WHOIS queries. The results are seen in Table 4.2. Nearly all the composite moduli were used in HTTPS or SMTP, but the HTTPS moduli were spread around the world while the SMTP moduli were only located in China. In HTTPS, North American and European countries were most heavily seen. The location spread in HTTPS and the relative

[1] `https://censys.io/`

[2] `https://github.com/zmap/zgrab`

[3] `https://github.com/zmap/zmap`

Label	# of IPs	Mod. Size (Bits)	Affected Protocols	Modulus
1	265	512	SMTP	`da583c16...4774e833`
2	242	2048	HTTPS	`c28992c5...d4681697`
3	28	4255	HTTPS	`4d494942...41674543`
4	5	1102	POP3S	`30818702...47020105`
5	2	1024	HTTPS	`a7790db6...288a9773`
6	2	1024	HTTPS	`cc17f2dc...8e073c6d`
7	2	2048	HTTPS	`8dd38f77...a8fdca8f`
8	1	904	HTTPS	`9ce85640...2220dc53`
9	1	1024	IMAPS, SMTP	`98ea99db...ab2b1b33`
10	1	1024	HTTPS	`d67de440...24218eb3`
11	1	2048	HTTPS	`f5a3da75...f564c113`
12	1	2048	SMTP, SMTPS	`ad85473c...3b2d764b`
13	1	4096	HTTPS	`9152ba0b...85fab358`

Table 4.1: **Composite DHE Moduli.** The frequency, affected protocols, and other properties of the composite DHE moduli used in the wild.

Affected Protocol	Number of IPs	Nationality
HTTPS	280	Austria, Bahrain, Bolivia, Canada, Chile, Czech Republic, France, Germany, India, Iraq, Israel, Italy, Japan, Lebanon, Malaysia, Mexico, Netherlands, Nicaragua, Pakistan, Poland, Romania, Saudi Arabia, Singapore, South Korea, Spain, Sweden, Taiwan, United States
IMAPS	1	Japan
POP3S	5	Ukraine
SMTP	267	China
SMTPS	1	Russia

Table 4.2: **Protocols and Countries.** Composite DHE moduli by protocol and country.

moduli abundance in SMTP increases the likelihood that these moduli are backdoors rather than random composites.

4.2.2 Composite Moduli Used By Web Servers

We first downloaded a Censys IPv4 scan to investigate DHE moduli in HTTPS. In April 2016, there were approximately 43M IP addresses in the HTTPS space, of which approximately 11M supported DHE. Over 300,000 distinct DHE moduli were observed across these 11M. We observed 5,783 unique non-safe prime moduli across 1.6M IPs, which will be further discussed in § 4.3.1. We observed 9 unique composite moduli across 280 IPs. We did a comparison to ECDHE and found that of 32 million IPs, all used a standard SECP curve, and that the server public ECDHE key was a valid point on the curve. This, of course, is consistent with expectation. Discovering composite DHE moduli, on the other hand, was not.

None of the composite moduli observed in HTTPS were export-grade; all were at least 904-bits in length. In May 2016, 46% of these IP addresses chose a Diffie-Hellman cipher suite by default, meaning forcing DHE (as described in § 4.4.1) is not needed in those cases.

To determine if these composite moduli were the result of a specific server implementation, we looked at the types of web servers using these moduli. The breakdown of these servers can be seen in Table 4.3. Apache servers were used by 125 IP addresses, which accounted for 45% of the IP addresses using composite moduli in HTTPS. Almost the same percentage of IP addresses (37%) did not specify a server. The remaining 21% of servers were spread over Microsoft, Oracle, Lighttpd, Nginx, and other servers specified by their company name.

Although Apache accounted for almost half the servers, the version numbers varied or did not exist. This trend was also seen in the other servers specified. Therefore the variety of servers and versions indicate that no one server implementation was responsible for the composite moduli.

The existence of composite moduli cannot be explained by poor entropy during generation, although poor entropy could potentially explain a systematic prime modulus. While it is possible that these composite moduli are pseudoprimes, enabling them to erroneously pass a probabilistic primality test, pseudoprimes occur so infrequently that they would not be a result of poor entropy. This fact coupled with the variety of server implementations means these moduli were potentially generated on purpose.

We then examined the public ownership information of the affected IPs in public databases and in the content of any public web pages. When the IP address owners and webpage content differed, both companies were considered identifiers for the IP address. For example, if one organization was supplied software by another, the second organization could have a logo displayed on the webpage. We decided to focus on companies associated with multiple IP addresses or with at least one active webpage. This left us with 21 companies: A1 Telekom Austria (A1), Amazon Web Services (AWS), Banco de Crédito (BCP), Bloomberg, Blue Coat Systems, Centre national de la recherche scientifique (CNRS), Deutsche Reisebüro (DER) Touristik, ELITE, Expedia, Eyou.net, FTSE Russell, JAMF Software, KDS, KPN, Nederlandse Spoorwegen (NS), NH Hotel Group, Nordea Bank, Santa Clara University (SCU), TravelTainment Germany, United Parcel Service (UPS), Universal Sompo General Insurance, and Universidad Nacional de Educación a Distancia (UNED).

We completed vulnerability disclosures to companies with at least one active webpage in HTTPS and which provided appropriate contact information; these disclosures are discussed in § 4.6. We also contacted the company with multiple affected IP addresses in SMTP. Companies in the tourism industry, such as TravelTainment and DER Touristik, accounted for about 50% of the IP addresses. The remaining companies were in various industries such as education and finance. Most companies, noticeably those with more affected IP addresses, had an active webpage.

To determine the longevity of composite moduli, we tested the 280 IP addresses three times during the course of writing to see if composite moduli were still used. In May 2016, 88% of the IP addresses still used the same composite modulus as before. Of the remaining 12% of IP addresses, about half switched to a prime modulus and half no longer connected under Diffie-Hellman. In June 2016, these statistics remained approximately constant. However, by August 2016, only 39% still used the same composite modulus and 53% used a prime modulus. The remaining 8% no longer connected under Diffie-Hellman, almost the same amount from May

Server	Number of Uses
Apache	95
Apache-Coyote/1.1	3
Apache/2.2.9 (Debian)	3
Apache/2.2.12 (Linux/SUSE)	1
Apache/2.2.15 (CentOS)	3
Apache/2.2.15 (Red Hat)	3
Apache/2.2.16 (Debian)	1
Apache/2.2.22 (Debian)	1
Apache/2.2.22 (Red Hat)	2
Apache/2.2.22 (Ubuntu)	2
Apache/2.4.3 (Unix)	3
httpd/1.00	8
Microsoft-IIS/7.5	2
Microsoft-IIS/8.0	1
Microsoft-IIS/8.5	6
Oracle Application Server 10g	1
Lighttpd	1
Nginx	24
Nginx/1.6.3	1
Nginx/1.9.10	1
Others	16
Not Specified	103

Table 4.3: **Web Servers.** Types of web servers using composite DHE moduli.

and June 2016. The decrease in composite moduli used could be attributed to our vulnerability disclosures and, independently, Wong's [85]. This assumption seemed to coincide with company responses, as many companies changed from composite moduli to prime as their primary response. Despite this, many composite moduli remained in use over months, indicating backdoored DHE parameters could go unnoticed for long periods of time.

4.2.3 Composite Moduli Used By Mail Servers

Since Censys did not have DHE scans for mail servers, we ran ZGrab scans in July 2016 on SMTP/S, POP3/S, and IMAP/S in TLS and STARTTLS looking for composite DHE moduli. We found 272 IP addresses with composite DHE moduli spread throughout IMAPS, POP3S, SMTPS, and SMTP. These results doubled the total number of composite moduli found, showing the problem extends beyond HTTPS.

IMAPS. Although there was only one IP address in IMAPS with a composite modulus, this IP address used the same modulus in SMTP. This modulus is number 9 in Table 4.1. The address is linked to a transportation company in Japan, which supports the trend of HTTPS companies that are not related to security and thus provide an advantageous attack target.

POP3S. There were five IP addresses in POP3S that all used the same composite modulus. This modulus is number 4 in Table 4.1. Although the company could not be determined accurately, the range of IP addresses suggested that only one Ukrainian company was involved.

SMTPS. Although there was only one IP address in SMTPS with a composite modulus, this IP address used the same modulus in SMTP. This modulus is number 13 in Table 4.1. This address is linked to a real estate company in Russia, which is also an industry that provides an advantageous attack target.

SMTP. Almost all the composite moduli in mail protocols were seen in SMTP. Out of 267 IP addresses with composite moduli, 265 used the same composite modulus (number 1 in Table 4.1). The remaining two were the IP addresses seen already in IMAPS and SMTPS. The 265 IP addresses were spread out across China, but all connected to an email service provider called Eyou.net [6]. This company was also contacted in the vulnerability disclosures described in § 4.6.

Popularity	Modulus (bits)	Subgroup (bits)	Source
76.9%	1024	160	MODP (RFC5114) [57]
11.3%	1024	160	Amazon Web Services
7.5%	768	160	`sun.security.provider`
3.2%	1024	160	`sun.security.provider`
0.3%	2048	224	MODP (RFC5114) [57]
0.1%	2048	224	`sun.security.provider`
~1%	–	–	(others)

Table 4.4: **Non-Safe Prime DHE Moduli.** The distribution and sources of non-safe DHE moduli.

4.3 Other DHE Parameter Investigation

This section outlines additional investigation into the areas of non-safe prime moduli, factorization of some moduli to recover significant portions of private DHE keys, and moduli used by open-source projects.

4.3.1 Non-Safe Prime Moduli Used By Web Servers

In addition to the composite moduli found in the HTTPS scan, we also found non-safe prime moduli used by 1.6M IPs. Of the 5,783 distinct non-safe primes we found, 5,409 were unique to a single IP. Six primes accounted for approximately 99% of sites. The distribution of non-safe primes is seen in Table 4.4. MODP groups were seen in 77% of IP addresses using non-safe primes. Parameters used in the `sun.security.provider` package by Java were seen in 11% of IPs using non-safe primes. This package has had previous instances of misconfigured Diffie-Hellman groups [8]. At the time of writing AWS load balancers no longer offer DHE cipher suites following a security policy update.

Safe prime groups have the property that all values in the range $1 < g < p-1$ are generators of groups of large order (either q or $2q$), and that an arbitrary value in this range is an element of $\mathbb{Z}_p^*$ with probability approaching $P = \frac{1}{2}$, meaning implementers are free to pick just about any generator they wish, and often opt for the smallest possible value (e.g. 2, 3, etc.). Non-safe prime groups, on the other hand, generally should be more select in their choice of generator, especially when the order of $\mathbb{Z}_p^*$ contains smooth factors. If a group element has an order containing smooth factors, partial recovery of the private DHE key is possible. For a random

```
11435638110073884015312138951374632602058078871389818 1372  \\
7578406924934826346123042770482700524507174581850431874 44  \\
984154616731278556112057558303927365079 55

= 5 * 11 * 3130497666273667404271 * 13239843891707982 4212  \\
370893794766672908033 * 501650748974370233413468006002745  \\
0130769436621955914589815397976412146715534764087911322 67
```

Figure 4.1: **512-bit Modulus Factorization.** Factorization of the 512-bit composite modulus found in SMTP.

non-safe prime group with an n-bit modulus and m-bit prime order subgroup $\mathbb{G}_q$, the probability an arbitrary value is a generator of $\mathbb{G}_q$ is approximately 2^{n-m}. Thus we should not generally expect to see generators such as 2 or 3 used in non-safe prime groups. We can expect such groups to leak more information about the exponent than the one bit of some safe prime groups.

Of the 1.6M IPs offering non-safe prime groups, we found 1,270 IPs using small generators. Generator values of 2 and 5 were most common but we also found cases of all prime numbers up to 31, as well as even values such as 4 and 6. This doesn't directly break DHE so long as (a) the order of the generator contains a large prime factor and (b) full-length exponents are used. This is a precarious situation, since the typical reason for using non-safe prime groups is precisely for the purpose of using short exponents (e.g. X9.42 groups [1]). It also speaks to the notion of parameter hygiene in which choices appropriate for one setting i.e. small generators of safe prime groups, is misapplied to another setting.

4.3.2 DHE Moduli Factorization

While a well-implemented DHE backdoor would *not* be exploitable, we set about conducting what partial factorizations of composite moduli we could. We used CADO-NFS and our own custom implementation of Pohlig-Hellman [68]/Pollard's P-1 [69] to recover, in many cases, numerous bits of a private DHE key. We factored the 512-bit composite SMTP modulus (number 1 in Table 4.1) revealing 5 factors seen in Figure 4.1.

We then factored $(f-1)$ of each factor f revealing the overall underlying group structure. The largest factor has a 280-bit subgroup, which prevented us from performing a complete discrete logarithm as the generator had order close to $p-1$. We were, however, able to recover 129 bits of the private DHE key using Pohlig-Hellman. The servers we examined appeared not to be using short exponents. If, however, a server did use a short exponent such as 160-bits, this SMTP prime *would* make an efficient backdoor: the first 129-bits could be recovered as described, and the remaining bits could be recovered from the 280-bit subgroup using Pollard's P-1 method in time approximately $2^{\frac{160-129}{2}} \approx 2^{16}$.

We conducted a partial factorization of the 904-bit composite modulus (number 8 in Ta-

$$\boxed{\begin{array}{l} 5 * 23 * 474289 * 726101 * 72240863 * 48794510505931 \\ * 70980749229449041 * 5093965413985867 * 2763354329179 \\ * 1711955530550801 * 71015949150893819 * \ldots \end{array}}$$

Figure 4.2: **904-bit Modulus Factorization.** Factorization of the 904-bit composite modulus found in HTTPS.

ble 4.1) and found a number of suspiciously smooth factors seen in Figure 4.2.

This site used an improper generator of 4, which allowed us similarly to recover 372 bits of the private DHE key. With either short exponents or knowledge of complete factorization, greater and more efficient recovery is possible.

Similar to with composite moduli, we also were able to conduct partial key recoveries in non-safe prime groups with improper generators. In one improper export-grade non-safe prime group we were able to recover a full half of the private DHE key (assuming a full-length exponent), though obviously for export-grade moduli, Logjam [8] would be a more efficient general attack strategy.

4.3.3 Survey of Open-source Projects

To determine if open-source projects use any weak moduli, we surveyed the default moduli of over a hundred open-source projects on GitHub. We used search terms based on common Diffie-Hellman byte array names (e.g., `dh1024_p`, etc.). Out of the 95 projects supporting export-grade 512-bit moduli, we found 16 distinct moduli, of which one was found in 44 projects. The most common modulus observed in Logjam was found in 9 projects. All were safe primes. Across 120 projects supporting 1024-bit moduli, there were 32 unique moduli. All the moduli were safe primes except for two: one reused from OpenSSL,[4] and a MODP group with 160-bit subgroup [57]. For 2048-bit moduli, there were 43 projects with 23 unique moduli. Similar to 1024-bit moduli, the only 2048-bit modulus that was not a safe prime was a MODP group with 256-bit subgroup [57]. For 3072-bit moduli, there were 3 unique safe primes spread over 4 projects. For 4096-bit moduli, there were 8 unique safe primes spread over 28 projects. Overall no weak moduli were found to be used, but parameter injection through an open-source project remains a possible attack vector for backdoors (see § 4.5.2).

4.4 Man-in-the-Middle Attack

This section discusses a man-in-the-middle (MITM) attack in TLS 1.2 and below to force DHE use, and explains the attack's limitations in SSH.

[4] `https://github.com/openssl/openssl/blob/master/test/ssltest_old.c`

4.4.1 Forcing DHE in TLS

As mentioned in § 2.4, cipher suites using DHE for key exchanges currently account for approximately 0.01-1% of TLS handshakes [65], limiting the potential for the attacker to exploit weak groups passively. Fortunately for the attacker – in this case active attacker Mallory – the message sequence of TLS makes it possible for someone knowing the master secret to actively modify the handshake to force DHE to be chosen if both parties support it. This is in contrast to SSH, which is not vulnerable to an active attack of this kind due to a differing message order (see § 4.4.2).

The client initiates a TLS handshake providing a list of supported cipher suites. Mallory modifies the client hello removing all but DHE cipher suites. The client and server exchange DHE keys as normal, except Mallory is able to exploit the weak or backdoored parameters to compute the discrete logarithm of the client or server public values and compute the pre-master secret g^{ab}, from which they can compute the master secret. With a careful choice of parameters Mallory can compute the discrete log in real-time. Finally using the master secret, Mallory forges fake client- and server-finished messages tricking the respective parties into believing the other party only supported DHE cipher suites, and thus there was no other choice but to connect under DHE. Furthermore, because the master secret is only a function of the pre-master secret and the client- and server-random values, both endpoints will derive the same master secrets, allowing the attacker – now passive attacker Eve – to continue *passively* eavesdropping the connection from this point forward. This attack is illustrated in Figure 4.3.

This MITM attack has some fundamental differences to the MITM attack proposed by Adrian et al. [8] (i.e. Logjam MITM). Our MITM forces DHE to be chosen by the server; the Logjam MITM forces the server to send export-grade DHE parameters to the client who believes non-export DHE is used. Both MITMs derive the master secret – while our MITM uses it to forge finished messages that allow the MITM to then *passively* eavesdrop on the TLS connection, the Logjam MITM uses the master secret to *actively* pretend to be the server.

4.4.2 Attack Limitations in SSH

The SSH protocol [87] specifies two fixed groups for Diffie-Hellman exchange: the 1024-bit Oakley group 2 [48] and the 2048-bit Oakley group 14 [54]. In major implementations of SSH, such as OpenSSH, these groups are included directly in the source code, although an extension of SSH does provide the option for a server to maintain its own list of group parameters [43]. Although the SSH standard calls specifically for the use of safe prime groups [43], older OpenSSH versions explicitly name non-safe primes as an option.[5]

[5] `http://man.openbsd.org/OpenBSD-4.3/cat5/moduli.0`

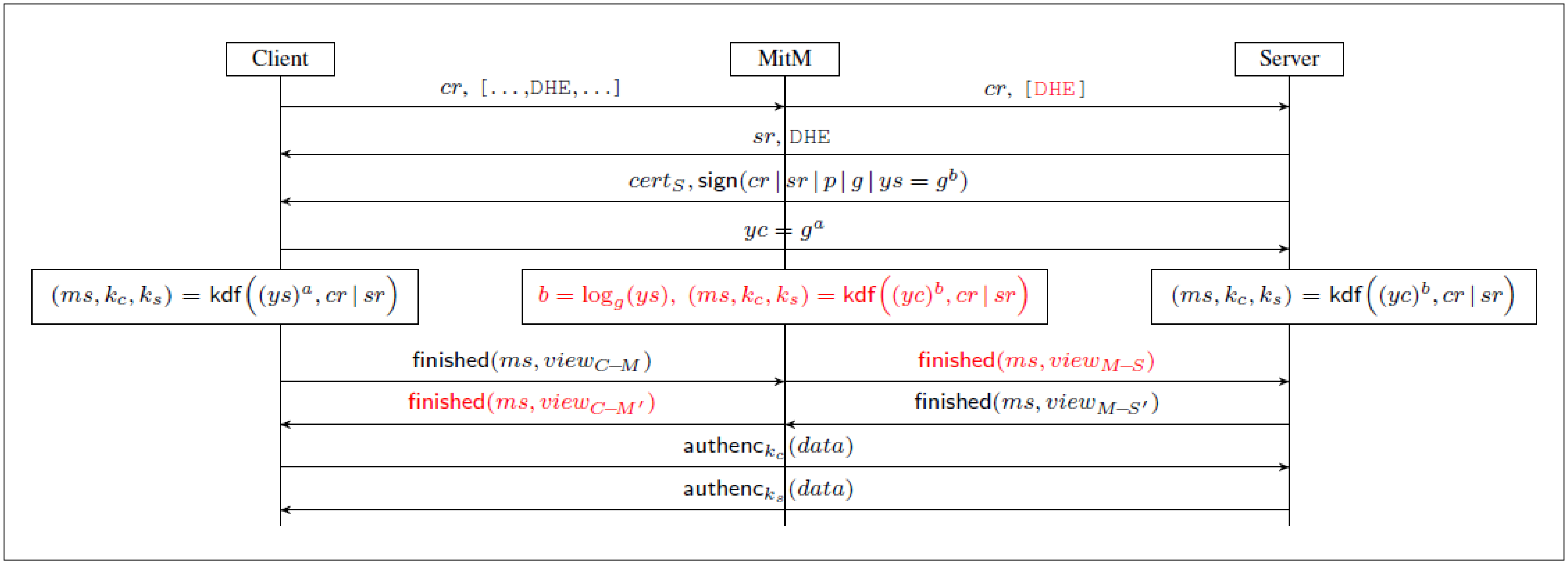

Client
MitM
Server
cr, [...,DHE,...]
cr, [DHE]
sr, DHE
$cert_S$, sign$(cr \mid sr \mid p \mid g \mid ys = g^b)$
$yc = g^a$
$(ms, k_c, k_s) = \mathsf{kdf}\big((ys)^a, cr \mid sr\big)$
$b = \log_g(ys)$, $(ms, k_c, k_s) = \mathsf{kdf}\big((yc)^b, cr \mid sr\big)$
$(ms, k_c, k_s) = \mathsf{kdf}\big((yc)^b, cr \mid sr\big)$
finished$(ms, view_{C\text{–}M})$
finished$(ms, view_{M\text{–}S})$
finished$(ms, view_{C\text{–}M'})$
finished$(ms, view_{M\text{–}S'})$
authenc$_{k_c}(data)$
authenc$_{k_s}(data)$

Figure 4.3: **Forcing DHE in TLS.** A man-in-the-middle with the ability to exploit weak or backdoored parameters can force the parties to select a DHE cipher suite against their natural preferences.

However in addition to SSH version restriction, active attacker Mallory would also have to force DHE during the connection. OpenSSH now prefers ECDHE for key exchange, so if Mallory wanted the parties to use DHE instead she would need to man-in-the-middle the handshake. Owing to the message sequence in SSH, being able to recover a DHE shared secret is not sufficient for this attack.

In SSH, the client chooses its preferred key-exchange method based on the server's indicated support [87]. Mallory could attempt to modify this initial server message, but then the attack would fail at the end of the handshake when the server provides a signed hash of the protocol messages. At this stage the client would detect that it saw a different sequence of messages than the server and would abort the connection, and Mallory could not forge this message without the server's private signing key. This is outside our threat model. If either party does not support ECDHE, but both parties support DHE, then they will connect under DHE.

4.5 Attack Vectors

The previous sections provided examples of potentially backdoored DHE moduli in the wild and discussed the subsequent implications. We now propose three scenarios that enable an attacker – in this case malicious designer Heidi – to position weak parameters for use as a backdoor. If the target uses these parameters to perform cryptographic operations (i.e. key generations, signatures, key agreements, encryptions, etc.), the associated security guarantees no longer hold. Since Diffie-Hellman group parameters are infrequently modified, attacking them can lead to *persistent* backdoors, even if the keys themselves are ephemeral. The proposed threat vectors include dropping the parameters onto a server, incorporating the parameters in an open-source project, and installing the parameters on a network appliance that ships to customers.

4.5.1 Attacking the Server

The most intuitive way to get backdoored parameters in use is to install them at the source. First, Heidi creates the weak parameters and chooses a target that supports Diffie-Hellman cipher suites. Second, Heidi injects these parameters as a backdoor payload onto the desired server. This step does require root access to the server, presumably in the context of a broader exploit. Having root access enables other attacks, such as stealing the server's private RSA signing key. This RSA attack would produce a similar outcome as the backdoored moduli, as efficient man-in-the-middle attacks are also possible for active attacker Mallory with the server's private RSA signing key. However, obtaining and using the private RSA key has two disadvantages. In many enterprise situations, the private RSA key is stored on a hardware security module (HSM) [7] attached to the server [25]. Since HSMs are designed to provide additional security to cryptographic keys, it would be difficult to steal a key stored on an HSM even with root access to the server. The second disadvantage to using the private RSA key is that it requires an active man-in-the-middle attack by Mallory. An active attack is also necessary to force DHE cipher suites when not preferred, but only during the handshake. However, as seen in § 4.2.2, *half* the IP addresses that use composite moduli in HTTPS prefer DHE cipher suites. Therefore Heidi could choose attack targets that prefer DHE cipher suites, allowing for passive eavesdropping by Eve instead of actively attacking with Mallory. This type of passive attack is only possible with backdoored moduli; using the private RSA signing key always requires an active attack.

Dropping the weak parameters onto the server requires no source code modification and creates a persistent backdoor; because of this, the backdoor may persist source code updates. The lack of parameter validation explained in § 3.2.1 and the examples of persistent composite moduli in § 4.2.2 mean that backdoored DHE moduli could remain undetected for some time.

4.5.2 Attacking the Application

The second threat scenario involves submitting the backdoored parameters to an open-source project rather than attacking the server directly. First, Heidi creates the weak parameters and finds an open-source project that supports Diffie-Hellman. Second, the parameters are submitted as a patch to that repository. Once the repository accepts the change, the persistent backdoor would then be installed for users of that project. Conversely, Heidi could create a new project that already contains the backdoored parameters. Since the Logjam disclosure, many GitHub projects have been updating their Diffie-Hellman parameters to remove 512-bit moduli and modify 1024-bit moduli. This widespread change could ironically provide a reason for Heidi to submit a patch.

Socat, an open-source data transfer relay, recently published a security advisory [74] that outlines a similar scenario, and was one of the motivations behind Wong's recent paper [85]. Here a hard-coded 1024-bit composite DHE modulus was discovered in the OpenSSL implementation. The Socat commit logs show that the composite modulus was introduced in January 2015 [73], and the security advisory was published more than a year later in February 2016, and the origin of the modulus remains unclear. Interestingly we also found this modulus twice in the HTTPS space (see modulus 6 in Table 4.1). This gap between implementation and detection indicates backdoored moduli could remain undetected for a long time. The individual associated with the commit deleted much of his Internet presence on the day the advisory was published [86]. Attempts to factor the modulus suggest that there are large factors, which could indicate a backdoor configuration such as those suggested in § 3.4. Although we didn't find any suspicious parameters in the GitHub projects mentioned in § 4.3.3, the Socat example suggests that starting a malicious open-source project is one potential delivery vector, and that the ad hoc nature of parameter checking would hinder detection.

4.5.3 Attacking the Network

The final threat scenario involves installing backdoored parameters onto a network appliance that is shipped to customers. Network appliances such as load balancers and traffic shapers are often used by companies to optimize application or network performance. Load balancers optimize application performance by distributing traffic across many servers, which decreases the load on individual servers. This traffic can be application or network traffic. Balancers also provide SSL termination so that servers do not have to perform encryption and decryption [5]. Although this invites man-in-the-middle attacks, the servers and balancer are often located on the same internal network which decreases this possibility. Another network appliance is traffic or packet shapers, which optimize network performance by delaying less important network packets. Various applications can be shaped differently, a process called application-based traffic shaping or deep packet inspection (DPI). Since DPI allows users to look at layers 2 through 7 of the OSI model, it is possible to view the ServerKeyExchange message [77]. DPI also provides the possibility of packet payload tampering [84].

This threat scenario requires Heidi to be a company employee who creates the weak parameters. Heidi then installs the backdoored parameters onto the load balancing network appliance sold by her company. Blue Coat's PacketShaper S-Series, a traffic shaping network appliance, can be connected with another PacketShaper to provide load balancing capability [4]. The load balancer equipped with backdoored parameters is then sold to a customer. The balancer sends decrypted traffic to the chosen server, then encrypts the server's response and sends it to the client as usual. Therefore the success of this scenario depends mostly on the trust placed in the load balancer to securely encrypt and decrypt traffic.

4.6 Vulnerability Disclosures

This section discusses the associations involved in publicly acknowledging vulnerabilities, and describes our vulnerability disclosures to companies along with their responses.

4.6.1 Public Acknowledgement of Vulnerabilities

The Common Vulnerabilities and Exposures (CVE) list[6] managed by the MITRE Corporation[7] provides publicly acknowledged vulnerabilities in information security, called CVE identifiers or CVEs informally, which are endorsed by the industry. A standardized set of vulnerability identifiers allows for easy reference and scoring by a multitude of systems, and removes interoperability issues stemming from a lack of standardization.

A vulnerability is acknowledged with a CVE identifier by a CVE Numbering Authority (CNA), primarily MITRE, and is placed in the publicly available list on the CVE website. A CVE identifier contains a number (e.g. CVE-2012-1723), a description such as affected products, and references such as security advisories.

The CVE list is also given to the U.S. National Vulnerability Database (NVD),[8] which provides additional information for each CVE such as a severity score. The Common Vulnerability Scoring System (CVSS) is an industry standard that provides a numerical severity score out of 10 and qualitative metrics for the vulnerability based on its exploitability and impact on systems. The primary standard for CVSS scoring is CVSS v2, although the current version of CVSS is CVSS v3 (released in 2015) and NVD reports both v2 and v3 scores.

4.6.2 Disclosure Methodology

As mentioned in § 4.2.2, we issued vulnerability disclosures to companies that were using composite moduli in HTTPS. Security contact information for each company was searched for in the HackerOne directory,[9] although only one company (Blue Coat Systems) had such information. Only companies with at least one active webpage were contacted, since webpage identifiers were important in determining the company associated with the IP address. Out of the 21 companies listed in § 4.2.2, only 17 were contacted. Only 47% of the contacted companies responded to our disclosure.

4.6.3 Disclosure to Blue Coat Systems

Blue Coat Systems, a billion-dollar company now owned by Symantec, was the first company contacted. We communicated on several occasions with a number of high-ranking employees within the company on the matter; in particular, we had multiple conference calls that included Blue Coat's Chief Technology Officer. A patch for the affected product, PacketShaper S-Series

[6] `https://cve.mitre.org/`

[7] `https://www.mitre.org/`

[8] `https://nvd.nist.gov/`

[9] `https://hackerone.com/directory`

11.5, was released in June 2016 along with a security advisory[10] acknowledging our contribution. A few weeks later on July 12, 2016, a CVE was released for this vulnerability under the label CVE-2016-5774 [3]. This CVE has a high severity score of 8.1 in CVSS v3 but only a medium score of 4.3 in CVSS v2, as v2 emphasizes percentage of impacted systems rather than level of impact as v3 does. Therefore although composite DHE moduli are not abundant in the wild, these moduli have a high degree of impact on affected systems. An interesting side effect of our disclosure was that it inadvertently uncovered a number of improperly configured web-facing administrator login pages, which allowed Blue Coat to follow up with affected customers.

4.6.4 Disclosure to Other Companies

After disclosure, the other 16 companies were split into three groups depending on the status of the vulnerability fix: completed, partially completed, or not started. At the time of writing, the vulnerability was fixed by 56% of these companies, although not all responded to us and three had implemented fixes prior to our disclosure. These independent solutions could have been a result of Wong's disclosures [85]. The solution implemented by most companies involved changing the composite moduli to prime, although one company simply removed its DHE cipher suites altogether. Of the 19% of companies who partially completed the vulnerability fix, all are progressively changing composite moduli to prime. The remaining 25% of companies did not respond to our disclosure and have not modified their Diffie-Hellman parameters. One of these companies had the highest number of affected IP addresses by far. A language barrier existed for some companies, which could have contributed to this result.

4.6.5 Company Responses

We spoke to senior management at Blue Coat and technical staff at many other companies. Despite this, all companies we had discussions with declined to provide us with information on the source of the potentially backdoored parameters. Blue Coat more specifically stated that the information could not be provided due to security reasons. Another company explained that its composite modulus was attributed to cipher modifications made by the company, but no specifics were given. Two others provided broad information on their load balancing, but not in the context of the specific vulnerability. As we were unable to receive external confirmation that these moduli were backdoored and could not completely factor the moduli to prove it, we cannot say unequivocally that these moduli are backdoored. We have discovered everything

[10] https://www.symantec.com/security-center/network-protection-security-advisories/SA127

possible about each company's vulnerability using publicly available information. Without additional information from the companies themselves, we cannot speculate further on topics such as the cause of the vulnerability.

4.7 Mitigation Strategies

There is a growing consensus that Diffie-Hellman negotiations are less secure than previously thought. Safari has removed DHE ciphersuites altogether, and Chrome plans to remove them in upcoming versions [18]. However, during the time of writing Chrome continued to offer DHE cipher suites if all other cipher suites offered were not accepted by the server. The current TLS 1.3 draft [72] proposes using named DHE groups [45], similar to the named ECDHE groups currently used. These named DHE groups are used in the `supported_groups` and `key_share` extensions, and would not be susceptible to the kinds of attacks described in this paper.

Information on using Diffie-Hellman properly has been extensively discussed by Adrian et al. [8], who suggest using at least 2048-bit Diffie-Hellman groups with safe prime moduli. Therefore we restrict our discussion to mitigation strategies for the outlined vulnerability. We propose four different strategies for mitigation: deprecating Diffie-Hellman cipher suites, verifying Diffie-Hellman parameters correctly, using named Diffie-Hellman groups, or modifying the ServerKeyExchange message to sign all previously seen messages.

Deprecate DHE. One option is to follow the example of Safari and Chrome and deprecate finite field Diffie-Hellman altogether. In our opinion, this option makes sense in certain situations, but not as a general solution. As we saw with Dual_EC_DRGB, there is a trade-off between trust and convenience through standardization. With that in mind, Bernstein et al. [19] added a new name to the standards of Alice and Bob: *Jerry*, an authority who generates curve parameters such that his attack cost is decreased. With the deprecation of RSA key exchange coming in TLS 1.3, DHE cipher suites represent the only alternative key exchange method.

Verify parameters properly. Our preferred option would be to simply implement the necessary domain parameter validation to begin with. The first issue, however, is computational cost. In order to verify that a generator or public DHE key has the intended order, modular exponentiation must be performed at runtime for *each* connection. Similarly p must be tested for primality, and, importantly, if general non-safe prime groups are to be permitted, the TLS and SSH protocols must provide an explicit means to communicate group order q. As we discussed in § 2.4.4, basic checking is not sufficient to prevent all attacks.

Use named parameters. A third solution is to develop standardized, named parameters similar to those in an ECC setting. The RFC proposed by Gillmor [45] and supported in the TLS 1.3 draft [72] involves standardizing parameters in the FFC setting to augment the MODP groups. As we see in ECC, named parameters are a feasible mitigation strategy used in the real world. One issue of restricting moduli to only safe primes is performance: private key lengths are 10 times larger than NIST recommended minimum standards. One performance optimization Gillmor suggests is to compromise by using safe prime groups with short, DSA-like exponents.

Change TLS. The last solution is to modify the `ServerKeyExchange` message so that all previously exchanged messages are also signed. The MITM attack from § 4.4.1 works as the `ServerKeyExchange` message only signs the DHE parameters, `ServerHello.random`, and `ClientHello.random`. If the list of cipher suites suggested in `ClientHello` and the chosen cipher suite in `ServerHello` were also signed, then the cipher suite tampering would be discovered upon receiving the `ServerKeyExchange` message. This solution was also proposed by Mavrogiannopoulos et al. [62] to prevent their cross-protocol attack.

Finally, a recent proposal by Bhargavan et al. [20] proposes an elegant method for downgrade resilience in TLS 1.3, and was incorporated into the draft as of Version 11. In their strategy, the server puts the highest version of TLS supported by the client into the `ServerHello.random`, which will be incorporated into the signed `ServerKeyExchange` message. If a client supports TLS 1.3, but is being man-in-the-middled in the context of a downgrade attack such the one described in § 4.5, the man in the middle will be unable to modify the signed `ServerKeyExchange` message, and the client will see that the server believes the client does not support TLS 1.3, which is false so the handshake is aborted. This method, combined with the use of named safe prime DHE groups in TLS 1.3, would solve the issue of backdoored groups.

Chapter 5

X.509 Certificate Name Mismatch Errors

5.1 Overview

In § 2.5.1, we explained that in TLS with Diffie-Hellman key exchange, an X.509 certificate attests to the ownership of the public key used to verify the signature on Diffie-Hellman parameters. One error that invalidates an X.509 certificate is a name mismatch error, as defined in § 2.5.3. Although there has been significant research on X.509 certificate errors in recent years, there has been less emphasis on name mismatch errors as studying them requires more than an IPv4 scan.

In this chapter, we conduct a survey of name mismatch errors based on scans of over 150 million domains. The domains are taken from the `.com`, `.info`, `.net`, and `.org` base domain sets. We implemented ZGrab, also used in Chapter 4, to obtain certificate data and found some disturbing results. We discovered that name mismatch errors occur in 69-79% of HTTPS connections, due largely to CDNs and hosting companies along with self-signed certificates. We further investigate HSTS-enabled websites and find that approximately 3% contain a name mismatch error that prevents their website from being accessed.

This chapter contains two sections: the methodology behind finding name mismatch errors, including related terminology, is discussed in § 5.2; and name mismatch error categorization along with the HSTS investigation is discussed in § 5.3.2.

5.2 Methodology

This section discusses the process for selecting a domain set and obtaining each domain's leaf certificate data in order to study the extent of name mismatch errors on the Internet.

5.2.1 Terminology

As discussed in § 2.5.3, a name mismatch error involves a mismatch between a website accessed over HTTPS and the names in the common name (CN) field and subject alternative name (SAN) extension of the website's certificate. In that section, we used the terms "domain" and "FQDN" briefly, but this chapter requires more specific definitions of "domain". Although we could define domains in terms of the Domain Name System (DNS) hierarchy, we instead focus on practical examples for each definition since they are considered in that context throughout the chapter.

Definition. (*Fully Qualified Domain Name.*) A Fully Qualified Domain Name (FQDN) (e.g. `www.example.com`), also known as an absolute domain, specifies an exact host on the Internet. For our purposes, the CN and SAN contain FQDNs.

Definition. (*Wildcard certificate.*) A wildcard certificate contains at least one FQDN in its CN or SAN that has an asterisk, *, in its far left position (e.g. `*.example.com`).

Definition. (*Top-Level Domain.*) A Top-Level Domain (TLD) (e.g. `.com`) is the portion of an FQDN located on the far right.

Definition. (*Base domain.*) A base domain (e.g. `example.com`) is the portion of an FQDN located directly left of the TLD and including the TLD itself.

Definition. (*Second-level domain.*) A second-level domain (e.g. `example`) is the portion of a base domain located to the left of the TLD.

Definition. (*Subdomain.*) A subdomain (e.g. `www.example.com`) is any domain that is a subset of another domain (e.g. `example.com`).

Definition. (*Zone file.*) For our purposes, we simplify the official definition of a zone file. We refer to a zone file as a list of all base domains registered to a specific TLD (e.g. all base domains for `.com`), although in actuality a zone file also contains additional DNS-related information.

Definition. (*Internal name.*) An internal name is part of a private network, such as the local area network (LAN) of an office. For our purposes, we are interested in internal names such as IP addresses and short names that are not FQDNs (e.g. `localhost`).

5.2.2 Domain Set Selection

Domains versus IP Addresses. To study name mismatch errors, an IPv4 scan similar to [52] and [37] is insufficient as the FQDN is needed to compare with the FQDNs in the CN and

SAN. A set of domains is required, and comprehensive domain sets can be found in zone files. Although zone files contain only base domains, they provide a list of every base domain registered to the specific TLD. This is in contrast to reverse DNS lookups (i.e. finding a FQDN from an IP address), which theoretically provide FQDNs [16] but in reality do not always provide accurate or indeed any results [52].

Zone File Selection. Similar to [75] and [83], we decided to use the zone files for `.com`, `.info`, `.net`, and `.org` to obtain a list of domains to study. These TLDs are generally considered the most popular and contain the most domains.[1] The `.com` TLD is particularly utilized; the `.com` zone file we obtained in May 2017 had 127 million domains compared to the next highest file, `.net`, which had 14 million. Although many zone files can be obtained by registering with the Centralized Zone Data Service (CZDS),[2] the zone files for `.com`, `.info`, `.net`, and `.org` can only be obtained by requesting access through their respective registries[3]. We also used the Alexa Top Million list,[4] which is routinely updated with the top million websites based on traffic volume. We attempted to get the zone file for `.ca` domains, but the Canadian Internet Registration Authority (CIRA)[5] does not allow access to this file.

5.2.3 Obtaining Name Mismatch Errors

Getting Leaf Certificates for Domains. After unique base domains are extracted from a zone file, the leaf certificate information (see § 2.5.2) for each domain needs to be found in order to find name mismatch errors. We used the DNS lookup tool ZDNS[6] to collect IP address(es) for each base domain, then used the associated application-layer scanner ZGrab (see § 4.2.1) to attempt TLS handshakes on port 443 (i.e. HTTPS) with each IP-domain combination. The X.509 leaf certificate information was extracted for successful handshakes. ZGrab supports SNI so the having multiple certificates on one IP address does not pose a problem. Domains were run with all their associated IP addresses for completeness, but any duplicated ZGrab results were removed after the scan so that only unique name mismatch errors were studied. We ran ZDNS+ZGrab on the cloud computing platform DigitalOcean.[7]

[1] Based on domain totals from `https://wwws.io/`

[2] `https://czds.icann.org/en`

[3] `.com` and `.net` are registered with `https://www.verisign.com/`, `.info` is registered with `https://afilias.info/`, and `.org` is registered with `https://pir.org/`

[4] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`

[5] `https://cira.ca/`

[6] `https://github.com/zmap/zdns`

[7] `https://www.digitalocean.com/`

Scans Completed. Two ZDNS+ZGrab scans were done on the `.com`, `.info`, `.net`, `.org`, and Alexa Top Million lists: one in May 2017 which scanned the base domains and saved the CN, and SAN if applicable, from the leaf certificate; and one in June 2017 which additionally scanned the `www` subdomains and also saved the self-signed boolean value. Since only the domain, CN, and optionally SAN are necessary to find a name mismatch error, only those values were saved to reduce result file size. The June 2017 scan additionally saved the self-signed boolean value to recognize self-signed leaf certificates. As explained in § 2.5.3, these certificates are considered invalid because they are signed by the certificate's subject, and tend to have additional validity issues beyond name mismatching.

Determining Name Mismatch Errors. The decision tree for determining name mismatch errors from certificate information is seen in Figure 5.1. The goal of the decision tree is to find name mismatch errors in two situations: the domain is a base domain, and does not match any of the FQDNs given in the CN and SAN (if applicable); or the domain is a subdomain (`www` of base domain, or other subdomain), and does not match any of the FQDNs given in the CN and SAN (if applicable) through either exact or wildcard matching. As an example of wildcard matching, `s1.s2.site.com` would match `*.s2.site.com` but not `*.site.com`.

We implemented the decision tree in Figure 5.1 in Python and ran it on the ZGrab results from the May and June 2017 scans. These results are discussed in § 5.3.

5.2.4 Potential False Positives and Negatives

The methodology described in § 5.2.3 has the possibility of giving false positives (i.e. outputs a name mismatch error when there is not one) and false negatives (i.e. outputs no error when there is one). We describe why these false results do not unduly affect our results, although they could be addressed in future work.

False Negative. In Figure 5.1, the name error decision tree checks the names in the CN even when the SAN extension is used. According to the HTTPS RFC [71], this is not allowed; the SAN only must be checked when it is used. Our results could include false negatives where a domain matched a name in the CN that was not included in the SAN. As our survey shows in § 5.3, name mismatch errors occur in approximately 75% of the HTTPS-enabled domains studied, so removing false negatives would only increase an already high percentage.

False Positive. It is possible to have an IP address support one HTTPS-enabled website and one or more HTTP websites. If a client tried to connect to the HTTP site using HTTPS, the certificate from the HTTPS-enabled website is fetched and a name mismatch error would

Set	Total Domains		Total Responding On Port 443 (%)		
	May 2017	June 2017	May 2017	June 2017 (Base)	June 2017 (`www`)
COM	126909094	127481013	31	31	33
INFO	5547030	5843252	20	19	20
NET	14910270	14939912	26	25	26
ORG	10426322	10402840	31	32	34
Alexa 1M	1000000	1000000	66	66	68

Table 5.1: **Domains Supporting HTTPS.** The percentage of domains supporting HTTPS from each domain set.

occur. This situation is technically a false positive since the requested website does not have a certificate (and so should not show an error), but since the client would consider this an error it does not change our results.

5.3 Name Mismatch Error Survey

This section discusses the name mismatch error results from the May and June 2017 scans described in § 5.2.3. The discussion includes the result difference over the two scans and categories of name mismatch errors such as having a name contain the domain's `www` subdomain. We additionally investigated domains on the HSTS preload list with name mismatch errors in July 2017.

5.3.1 Percentage of Domains with Name Mismatch Errors

To determine if name errors across TLDs persisted across time and subdomains, we investigated the percentage of name errors in the May 2017 base domains, June 2017 base domains, and June 2017 `www` subdomains. Table 5.1 shows the number of domains tested in each scan along with the percentage that responded on HTTPS. Both the number of domains and the amount that responded on port 443 (i.e. HTTPS) remained approximately constant. It was reassuring to see that the most popular websites from the Alexa Top Million had a higher propensity to use HTTPS.

We next investigated the percentage of HTTPS-enabled domains that had a name mismatch with the CN and SAN names on their certificates, seen in Table 5.2. Overall, there was a disturbingly high percentage of name mismatch errors seen across all zone file domains and a

Set	Total with Name Mismatch (% of HTTPS Domains)		
	May 2017	June 2017 (Base)	June 2017 (`www`)
COM	74	71	69
INFO	79	77	77
NET	78	76	75
ORG	74	73	71
Alexa 1M	37	36	37

Table 5.2: **Domains With Name Mismatch Errors.** The percentage of HTTPS-enabled domains with a name mismatch error.

negligible difference between the base domain and `www` subdomain results. Name mismatch errors occurred for 69-79% of domains responding to HTTPS, which is significantly higher than 20% found by Akhawe et al. [9] but slightly less than Ristic et al. [75]. Although the name mismatch error percentages did decrease between May and June, it was small compared to the number of domains affected, showing that the initial results painted an accurate picture. The Alexa Top Million websites were less affected than the zone file domains, but considering these websites are accessed the most frequently of all, having one third affected is still concerning. In the next section, we set out to categorize the errors to determine the main causes behind their frequent occurrence.

5.3.2 Categories of Domains with Name Mismatch Errors

The name mismatch errors for each domain set were separated into eight categories in order, where a name error was put only into the first category it matched. The breakdown for each scan is seen in Tables table 5.3, table 5.4, and table 5.5.

(1) **Self-signed Certificates (June 2017 only).** Self-signed certificates are issued and signed by the same entity, which for leaf certificates means the website itself. These certificates usually have additional issues because they have not been screened by a CA, and therefore are removed from the name mismatch list first.

In the June 2017 scans, self-signed certificates accounted for between 15.6-20.3% of name mismatch errors from zone file domains. This result is consistent with Akhawe et al. [9], who found that 3% of invalid connections were due to self-signed certificates and 19% were due to name mismatch errors. Holz et al. [52] and Durumeric et al. [37] found

Category	Percentage of Name Mismatch Errors (%)				
	COM	INFO	NET	ORG	Alexa 1M
Self-signed	-	-	-	-	-
CDNs	46.7	41.8	43.8	47.1	34.8
`www` Subdomain	0.2	0.1	0.3	0.3	5.4
Base Domain	-	-	-	-	0
Other Subdomain	0.3	0.3	0.6	0.4	5.2
Longest Domain Piece	0.3	2.3	2.0	1.5	2.1
IP Address	0.2	0.3	0.3	0.4	1.3
No Dots	6.3	8.2	7.0	5.3	12.2
Undefined	46	47	46	45	39

Table 5.3: **Name Mismatch Errors Categorization, May 2017 (Base).** The percentage of name mismatch errors from the May 2017 scan of base domains that could be categorized.

Category	Percentage of Name Mismatch Errors (%)				
	COM	INFO	NET	ORG	Alexa 1M
Self-signed	15.8	20.3	17.6	16.5	26.3
CDNs	46.2	42.0	45.7	47.1	34.2
`www` Subdomain	0.3	0.1	0.2	0.3	5.3
Base Domain	-	-	-	-	≈ 0
Other Subdomain	0.2	0.2	0.4	0.3	4.1
Longest Domain Piece	0.2	2.2	1.8	1.4	1.9
IP Address	0.1	0.1	0.1	0.3	≈ 0
No Dots	0.2	0.1	0.2	0.1	0.2
Undefined	37	35	36	34	28

Table 5.4: **Name Mismatch Errors Categorization, June 2017 (Base).** The percentage of name mismatch errors from the June 2017 scan of base domains that could be categorized.

Category	Percentage of Name Mismatch Errors (%)				
	COM	INFO	NET	ORG	Alexa 1M
Self-signed	15.6	19.8	17.4	16.2	24.0
CDNs	46.2	41.1	43.4	47.2	42.5
`www` Subdomain	-	-	-	-	-
Base Domain	0.4	0.4	0.6	0.5	3.0
Other Subdomain	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
Longest Domain Piece	0.5	2.5	2.3	1.7	4.3
IP Address	0.1	0.1	0.1	0.3	≈ 0
No Dots	0.2	0.1	0.2	0.1	0.2
Undefined	37	36	36	34	26

Table 5.5: **Name Mismatch Errors Categorization, June 2017 (`www`).** The percentage of name mismatch errors from the June 2017 scan of `www` subdomains that could be categorized.

twice the number of self-signed certificates, but they checked for self-signed certificates over all certificate errors instead of only mismatch errors.

From our results, it was concerning to see that Alexa Top Million domains had a larger percentage of self-signed certificates than those from the zone files. Since these domains are the most heavily visited, they should be more conscious about security, but this is not the case based on the percentage of self-signed certificates.

(2) **Web Hosting Companies and CDNs.** Content delivery networks (CDNs), web hosting companies, and other companies that contain others' website information on their servers are known to frequently configure TLS incorrectly [58, 31, 30]. With this idea in mind, we identified over 200 CDNs and related companies based on the CN and SAN names of domains with name mismatch errors. The full list can be seen in Appendix B. It includes companies from Canada, the United States, Japan, and Russia among other countries. More important than the specific companies is the widespread adoption of careless CN and SAN selections; many name mismatch errors remain unidentified. The 200 companies we found were only some of the possible companies, as analysing the full set of name errors by companies was too time intensive to complete. Therefore, many of the undefined name errors included additional companies, further emphasizing the widespread mismatch errors.

In general, name mismatch errors attributed to a CDN or other company made up over 40% of the name errors for both zone file domains and Alexa domains. This result is somewhat confirmed by Holz et al. [52], who find that 48% of invalid certificates in XMPPS related to the CDN `incapsula.com`. In contrast to the self-signed certificate results, the Alexa domains had a smaller percentage for its base domain scans than the zone file domains. Some of the more frequently occurring names in the CN or SAN included `websitewelcome.com` and variations of `hostgator` (i.e. HostGator), `secureserver.net` (i.e. GoDaddy), `xserver.jp`, variations of `akamai` (i.e. Akamai), `weebly.com`, `sakura.ne.jp`, `wpengine.com` (i.e. WordPress), `webhostbox.net` (i.e. ResellerClub), `bluehost.com`, and `kasserver.com` (i.e. Mertens Media).

(3) **www Subdomain.** For scans of the base domains, we checked if its `www` subdomain was present in the CN or SAN. This result was less than 1% for the zone file domains but around 5% for the Alexa domains, both making up only a small portion of the overall name errors. The higher percentage in the Alexa domains could indicate a better attempt at certificate validity, as a name mismatch error due to a subdomain is less blatant than a CDN name for example.

(4) **Base Domain.** For the scan of the `www` subdomains, we checked if its base domain was present in the CN or SAN. Similar to `www` subdomain matching, this category constituted only a small portion of name errors and was slightly higher for the Alexa domains, again indicating a better attempt at name matching.

(5) **Other Subdomain.** For scans of the base domains and `www` subdomains, we checked if that domain was present within another name in the CN or SAN (e.g. a subdomain besides the `www` version). For the base domain scans, this category was approximately similar to both `www` subdomain matching and base domain matching. However, for the `www` subdomain scan, this category was negligible, a result which is expected as subdomains containing a `www` subdomain are not commonly seen.

(6) **Longest Domain Piece.** For scans of the base domains and `www` subdomains, we checked if the longest piece from that domain was present within another name in the CN or SAN. Ideally, this piece was the second-level domain (e.g. `example` from `example.com`) so that it described the website, but it could also point to a different piece [9] (e.g. `example` from `example.site.com`). This category generally consisted of less than 3% of name mismatch errors, and was present slightly more frequently in the `.info` and Alexa domains.

(7) **IP Address.** For scans of the base domains and www subdomains, we checked if an IP address was one of the names in the CN or SAN. Certificates are prohibited from having internal names such as IP addresses as of November 2015, and previously existing certificates should have been revoked by October 2016 [26]. This standard was created to prevent MITM attacks that take advantage of non-unique internal names; a MITM could request a certificate with the same internal name as its target. We found IP addresses in small percentages, but seeing any IP address is concerning since all certificates containing them should have been revoked months ago.

(8) **No Dots.** For scans of the base domains and www subdomains, we checked if a name without dots (i.e. not an FQDN) was one of the names in the CN or SAN. This method could miss names that are not FQDNs but have dots (e.g. `localhost.localdomain`). The names in this category are internal names, and therefore certificates containing them should have been revoked [26]. In June 2017, there were relatively few found, but there was many more in May 2017; the zone domains and Alexa domains had over 5% of their name mismatch errors because of this category. It is possible that the revocation of these certificates was delayed and occurred between May and June 2017. Regardless, the small presence in June 2017 indicates that certificates containing names without dots still exist.

5.3.3 HSTS Domains with Name Mismatch Errors

As explained in § 2.2.4, websites can use the HTTP Strict Transport Security (HSTS) mechanism to specify that they can only be accessed over HTTPS. A website using HSTS but having an invalid certificate will be "locked"; HSTS prevents users from continuing to such websites. In this section, we investigated invalid certificates in the context of name mismatch errors for websites on Chrome's HSTS preload list[8] since other browsers' lists are based off this list [76]. The list contains additional information beyond HSTS entries, but that information is not relevant to this work.

We obtained Chrome's HSTS preload list in July 2017, and extracted FQDNs that supported HSTS from the list. After adding the www subdomains from websites that supported HSTS for their subdomains, we had 57258 FQDNs. Using the methodology described in § 5.2.3, in July 2017 we ran ZDNS+ZGrab to attempt TLS handshakes with each domain and get the certificate's CN and SAN if possible. Of the FQDNs, only 82% responded on port 443 (HTTPS). The remaining 18% could be websites that are no longer active or that are awaiting

[8] `https://cs.chromium.org/chromium/src/net/http/transport_security_state_static.json`

exclusion from the list, since it takes months[9] for a removal to propagate.

Out of the FQDNs that responded to HTTPS, there were 1320 (2.8%) that had a name mismatch error. These websites would show a certificate error when accessed from a browser and HSTS would prevent the user from continuing to the website. We tested a random sampling of these FQDNs in Chrome which confirmed this behaviour. We searched the domains for websites that would be relevant to a variety of users, such as government or banking websites, and found a few examples. However, a Google search of those websites showed that none of them were utilized – the utilized website had a valid certificate and was a www subdomain or base domain of the erroneous website, or in some cases was a different website entirely. As an example, ncpc.gov is a United States government site, and while it had a name mismatch error, the utilized website was actually www.ncpc.gov which had a valid certificate. Another example is ebankcbt.com, a banking website, where the utilized website www.gocitizens.bank had a valid certificate. Even Google had a name mismatch error for www.groups.google.com, although the utilized website was actually groups.google.com.

Although we only found instances of name mismatch errors for websites that had a properly configured website on another name, we argue that this practice sets a bad precedent. There were less than 60000 websites that supported HSTS on Chrome's preload list, which is almost negligible compared to the hundreds of millions of domains we examined in § 5.3. Although we did not investigate the Strict-Transport-Security header, potentially missing HSTS websites, neither did we investigate other errors such as self-signed certificates that could "lock out" additional websites. The websites on the HSTS preload list have an obligation to set a standard – it is bad practice to force HTTPS but use invalid certificates – and the presence of name mismatch errors does not inspire confidence.

[9] According to https://hstspreload.org/

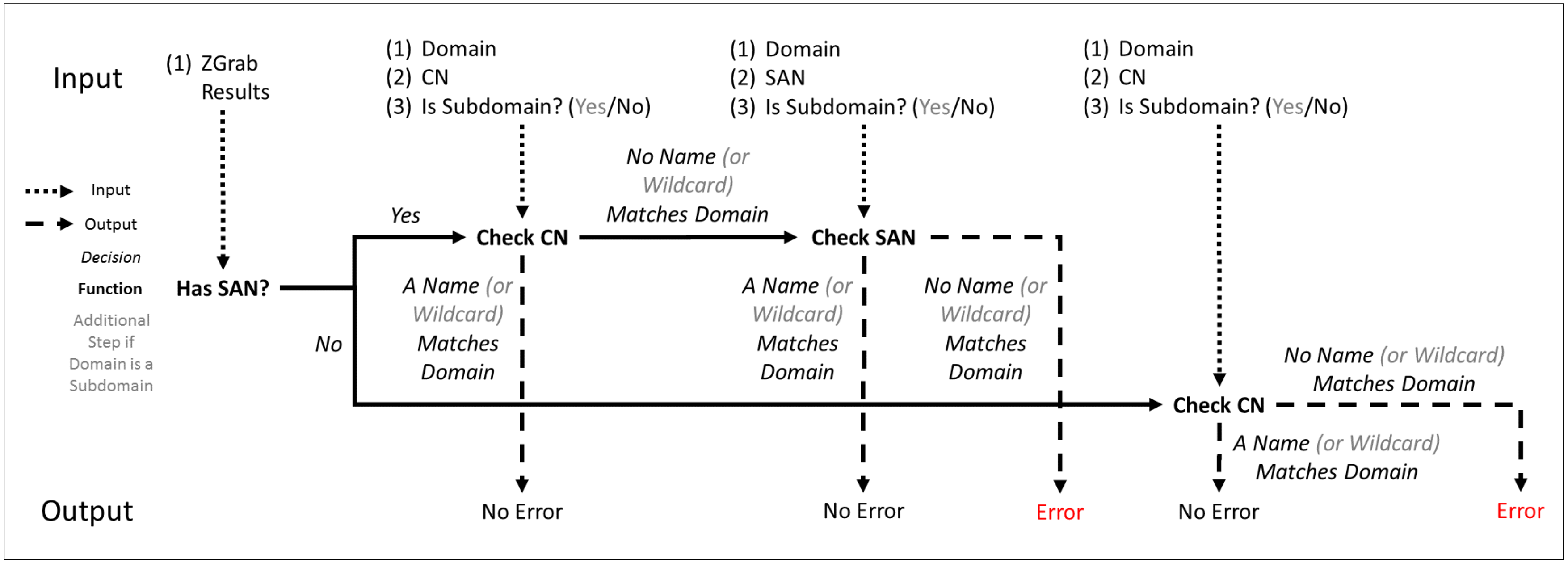

Input
(1) ZGrab Results
(1) Domain
(2) CN
(3) Is Subdomain? (Yes/No)
(1) Domain
(2) SAN
(3) Is Subdomain? (Yes/No)
(1) Domain
(2) CN
(3) Is Subdomain? (Yes/No)
Input
Output
Decision
Function
Additional Step if Domain is a Subdomain
Has SAN?
Yes
No
Check CN
No Name (or Wildcard) Matches Domain
Check SAN
A Name (or Wildcard) Matches Domain
A Name (or Wildcard) Matches Domain
No Name (or Wildcard) Matches Domain
Check CN
No Name (or Wildcard) Matches Domain
A Name (or Wildcard) Matches Domain
Output
No Error
No Error
Error
No Error
Error

Figure 5.1: **Name Error Decision Tree.** Name mismatch errors were decided based on the input domain's structure.

Chapter 6

Conclusion and Future Work

In this thesis, we discovered a new vulnerability in the implementation of the Diffie-Hellman key exchange. Poor Diffie-Hellman parameter validation enabled DHE implementations to connect under weak and potentially backdoored parameters, which we demonstrated in all major browsers. We proposed a Diffie-Hellman backdoor construction that would allow an attacker to efficiently compute the discrete logarithm while denying the backdoor's existence. We then conducted a survey of DHE parameters across TLS and STARTTLS and found hundreds of potentially backdoored parameters in use. A large portion of the private DHE key was recovered for some of these parameters. DHE cipher suites account for a small number of TLS connections but are still well supported, so we proposed a man-in-the-middle attack to force DHE use by an attacker exploiting a backdoor. Vulnerability disclosures were completed for 17 companies, and in the most significant case we had several conference calls with the CTO of a billion-dollar company that resulted in a publicly acknowledged vulnerability.

We additionally conducted a survey on name mismatch errors in HTTPS for over 150 million websites, and found that on average 75% of HTTPS connections are invalidated by name mismatch errors. After categorizing these errors, we determined that at least 40% were caused by invalid certificates owned by web hosting or content delivery network companies. We also found over 1000 websites that force HTTPS use but have a name mismatch error, making them inaccessible.

Our work on Diffie-Hellman adds to many related works, and together we have significantly decreased support for DHE cipher suites. In 2015, Adrian et al. [8] implemented a downgrade attack that would allow 512-bit Diffie-Hellman parameters to be used, and employed precomputation to recover the private DHE key. In 2016, Bhargavan et al. [20] proposed downgrade protection, incorporated into the TLS 1.3 draft [72], which prevents downgrades to DHE cipher suites. In 2017, concurrent but independent work by Valenta et al. [81] also investigated the exploitation of weak DHE parameters through lack of parameter validation. This combined work has decreased support for Diffie-Hellman significantly – in the time between writing our paper [35] and this thesis, the most widely used browser (Google Chrome) has removed DHE cipher suites, and telemetry data from Mozilla Firefox [65] indicates that default DHE connections have decreased from 1% to almost 0%.

Our work on name mismatch errors uncovered startling statistics on the prevalence of invalid certificates in use. The methodology was sound overall, but had some slight flaws that could be improved in future work. While we were able to categorize the likely reason behind many of the errors, on average 40% remained undefined due to the high number of errors. We speculate that many of these undefined errors are also due to web hosting and content delivery networks, but due to time constraints we were only able to identify 200 companies. It would be interesting to fully investigate the name mismatch errors and track any changes over a longer time period – as shown by our Diffie-Hellman scans, sometimes the most interesting findings are hidden among data sets of millions.

Bibliography

[1] "Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography," American National Standards Institute (ANSI), Tech. Rep., 2003.

[2] "CVE-2016-0701 Detail," 2016, https://nvd.nist.gov/vuln/detail/CVE-2016-0701.

[3] "CVE-2016-5774 Detail," 2016, https://nvd.nist.gov/vuln/detail/CVE-2016-5774.

[4] "Standby Feature with High Availability Clusters," 2016, https://bto.bluecoat.com/packetguide/11.6/Content/PDFs/standby.pdf.

[5] "What is an SSL Load Balancer?" 2016, https://www.nginx.com/resources/glossary/ssl-load-balancer/.

[6] "Company Overview of Eyou.net," 2017, http://www.bloomberg.com/research/stocks/private/snapshot.asp?privcapId=113374953.

[7] "Hardware security module," 2017, https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.5.0/com.ibm.dp.doc/hsm2.html.

[8] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[9] D. Akhawe, J. Amann, M. Vallentin, and R. Sommer, "Here's My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web," in *International World Wide Web Conference*, May 2013.

[10] D. Akhawe and A. P. Felt, "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness," in *22nd USENIX Security Symposium*, Aug. 2013.

[11] R. Anderson and S. Vaudenay, "Minding Your P's and Q's," in *ASIACRYPT*, 1996, pp. 26–35.

[12] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS with SSLv2," in *25th USENIX Security Symposium*, Aug. 2016.

[13] E. Barker, L. Chen, A. Roginsky, and M. Smid, “Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography,” National Institute of Standards and Technology (NIST), Tech. Rep., 2013.

[14] E. Barker and A. Roginsky, “Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths,” National Institute of Standards and Technology (NIST), Tech. Rep., 2015.

[15] R. Barnes, M. Thomson, A. Pironti, and A. Langley, “Deprecating Secure Sockets Layer Version 3.0,” Jun. 2015, https://tools.ietf.org/html/rfc7568.

[16] D. Barr, “Common DNS Operational and Configuration Errors,” Feb. 1996, https://tools.ietf.org/html/rfc1912.

[17] M. Benantar, *Access Control Systems: Security, Identity Management and Trust Models*. Springer Science and Business Media, 2006.

[18] D. Benjamin, “Intent to Remove: DHE-based ciphers,” 2016, https://groups.google.com/a/chromium.org/forum/#!topic/security-dev/sVq6r0i-CZM.

[19] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, E. Lambooij, T. Lange, R. Niederhagen, and C. van Vredendaal, “How to manipulate curve standards: a white paper for the black hat,” 2014, http://bada55.cr.yp.to/bada55-20150927.pdf.

[20] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, , M. Kohlweiss, and S. Zanella-Béguelin, “Downgrade Resilience in Key-Exchange Protocols,” in *IEEE Symposium on Security and Privacy*, 2016.

[21] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub, “Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS,” in *IEEE Symposium on Security and Privacy*, May 2014.

[22] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti, “Verified Contributive Channel Bindings for Compound Authentication,” in *Network and Distributed System Security Symposium (NDSS ’15)*, Feb. 2015.

[23] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS),” May 2006, https://tools.ietf.org/html/rfc4492.

[24] D. Boneh, A. Joux, and P. Q. Nguyen, “Why Textbook ElGamal and RSA Encryption Are Insecure,” in *ASIACRYPT*, 2000, pp. 30–43.

[25] K. Cairns, J. Mattsson, R. Skog, and D. Migault, “Session Key Interface (SKI) for TLS and DTLS,” Oct. 2015, https://tools.ietf.org/html/draft-cairns-tls-session-key-interface-01.

[26] “Guidance on the Deprecation of Internal Server Names and Reserved IP Addresses,” Certificate Authorities/Browser Forum, 2016, https://cabforum.org/internal-names/.

[27] N. Chang-Fong and A. Essex, "The Cloudier Side of Cryptographic End-to-end Verifiable Voting: A Security Analysis of Helios," in *Annual Computer Security Applications Conference (ACSAC '16)*, Dec. 2016.

[28] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and T. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," May 2008, https://tools.ietf.org/html/rfc5280.

[29] J.-S. Coron, A. Joux, A. Mandal, D. Naccache, and M. Tibouchi, "Cryptanalysis of the rsa subgroup assumption from tcc 2005," in *14th International Conference on Practice and Theory in Public Key Cryptography (PKC)*, D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, Eds., 2011, pp. 147–155.

[30] C. Culnane, M. Eldridge, A. Essex, and V. Teague, "Trust Implications of DDoS Protection in Online Elections," in *International Conference on E-Voting and Identity (E-Vote-ID)*, 2017, to appear.

[31] C. Culnane, M. Eldridge, A. Essex, V. Teague, and Y. Yarom, "iVote West Australia: Who voted for you?" Mar. 2017, https://pursuit.unimelb.edu.au/articles/ivote-west-australia-who-voted-for-you.

[32] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," Jan. 1999, https://tools.ietf.org/html/rfc2246.

[33] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Aug. 2008, https://tools.ietf.org/html/rfc5246.

[34] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[35] K. Dorey, N. Chang-Fong, and A. Essex, "Indiscreet Logs: Diffie-Hellman Backdoors in TLS," in *Network and Distributed System Security Symposium (NDSS '17)*, Feb. 2017.

[36] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A search engine backed by Internet-wide scanning," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[37] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS Certificate Ecosystem," in *Internet Measurement Conference (IMC '13)*, Oct. 2013.

[38] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Internet Measurement Conference (IMC '14)*, Nov. 2014.

[39] D. Eastlake, "Transport Layer Security (TLS) Extensions: Extension Definitions," Jan. 2011, https://tools.ietf.org/html/rfc6066.

[40] P. Eckersley and J. Burns, "An Observatory for the SSLiverse," in *DEFCON 18*, Jul. 2010.

[41] A. P. Felt, R. W. Reeder, H. Almuhimedi, and S. Consolvo, “Experimenting At Scale With Google Chrome’s SSL Warning,” in *ACM CHI Conference on Human Factors in Computing Systems*, 2014.

[42] J. Fried, P. Gaudry, N. Heninger, and E. Thomé, “A kilobit hidden SNFS discrete logarithm computation,” in *EUROCRYPT*, 2017.

[43] M. Friedl, N. Provos, and W. A. Simpson, “Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol,” Mar. 2006, https://tools.ietf.org/html/rfc4419.

[44] T. Gigler, M. Coates, D. Wichers, T. Reguly, and T. Hsu, “Transport Layer Protection Cheat Sheet,” Apr. 2017, https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet.

[45] D. K. Gillmor, “Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS),” Aug. 2016, https://tools.ietf.org/html/rfc7919.

[46] D. M. Gordon, “Designing and detecting trapdoors for discrete log cryptosystems,” in *ADVANCES IN CRYPTOLOGY– CRYPTO ’92*. Springer-Verlag, 1993, pp. 66–75.

[47] J. Groth, “Cryptography in subgroups of z*n,” in *Theory of Cryptography Conference (TCC)*, 2005.

[48] D. Harkins and D. Carrel, “The Internet Key Exchange (IKE),” Nov. 1998, https://tools.ietf.org/html/rfc2409.

[49] R. Henry and I. Goldberg, “Solving discrete logarithms in smooth-order groups with CUDA,” in *SHARCS*, 2012.

[50] J. Hodges, C. Jackson, and A. Barth, “HTTP Strict Transport Security (HSTS),” Nov. 2012, https://tools.ietf.org/html/rfc6797.

[51] P. Hoffman, “SMTP Service Extension for Secure SMTP over Transport Layer Security,” Feb. 2002, https://tools.ietf.org/html/rfc3207.

[52] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar, “TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication,” in *Network and Distributed System Security Symposium (NDSS ’16)*, Feb. 2016.

[53] 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 *Internet Measurement Conference (IMC ’11)*, Nov. 2011.

[54] T. Kivinen and M. Kojo, “More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE),” May 2003, https://tools.ietf.org/html/rfc3526.

[55] M. Kranch and J. Bonneau, “Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning,” in *Network and Distributed System Security Symposium (NDSS ’15)*, Feb. 2015.

[56] A. Lenstra, “Constructing trapdoor primes for the proposed DSS,” École polytechnique fédérale de Lausanne, Tech. Rep., 1991.

[57] M. Lepinski and S. Kent, “Additional Diffie-Hellman Groups for Use with IETF Standards,” Jan. 2008, https://tools.ietf.org/html/rfc5114.

[58] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, “When HTTPS Meets CDN: A Case of Authentication in Delegated Service,” in *IEEE Symposium on Security and Privacy*, May 2014.

[59] C. H. Lim and P. J. Lee, “A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup,” *Crypto*, vol. 1294, pp. 249–263, 1997.

[60] M. Marlinspike, “More Tricks for Defeating SSL in Practice,” in *Black Hat USA*, 2009.

[61] N. Mavrogiannopoulos, “Using OpenPGP Keys for Transport Layer Security (TLS) Authentication,” Nov. 2007, https://tools.ietf.org/html/rfc5081.

[62] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, “A cross-protocol attack on the TLS protocol,” in *ACM Conference on Computer and Communications Security*, Oct. 2012, pp. 62–72.

[63] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[64] B. Möller, T. Duong, and K. Kotowicz, “This POODLE Bites: Exploiting The SSL 3.0 Fallback,” Google Security Advisory, 2014, https://www.openssl.org/~bodo/ssl-poodle.pdf.

[65] “Telemetry Dashboards - Measurement Dashboard,” Mozilla, 2017, https://telemetry.mozilla.org/.

[66] E. Nygren, “Reaching toward universal TLS SNI,” Mar. 2017, https://blogs.akamai.com/2017/03/reaching-toward-universal-tls-sni.html.

[67] R. Oppliger, *SSL and TLS: Theory and Practice*. Artech House, 2009.

[68] S. C. Pohlig and M. E. Hellman, “An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance,” *IEEE Transactions on Information Theory*, vol. 24, no. 1, pp. 106–110, 1978.

[69] J. M. Pollard, “Theorems on Factorization and Primality Testing,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 76, no. 3, pp. 521–528, 1974.

[70] ——, “Monte Carlo Methods for Index Computation (mod p),” *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, 1978.

[71] E. Rescorla, “HTTP over TLS,” May 2000, https://tools.ietf.org/html/rfc2818.

[72] ——, “The Transport Layer Security (TLS) Protocol Version 1.3,” Jul. 2017, https://tools.ietf.org/html/draft-ietf-tls-tls13-21.

[73] G. Rieger, “FIPS requires 1024 bit DH prime,” 2015, http://repo.or.cz/socat.git/commitdiff/281d1bd6515c2f0f8984fc168fb3d3b91c20bdc0.

[74] ——, “Socat security advisory 7 - Created new 2048bit DH modulus,” 2016, http://www.openwall.com/lists/oss-security/2016/02/01/4.

[75] I. Ristić, “Internet SSL Survey 2010,” in *Black Hat USA*, Jul. 2010.

[76] ——, *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, 2014.

[77] W. G. Sanchez, “SLOTH Downgrades TLS 1.2 Encrypted Channels,” 2016, http://blog.trendmicro.com/trendlabs-security-intelligence/sloth-downgrades-tls-1-2-encrypted-channels/.

[78] Y. Sheffer, R. Holz, and P. Saint-Andre, “Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS),” May 2015, https://tools.ietf.org/html/rfc7525.

[79] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th ed. Pearson Education, 2017.

[80] G. Surman, “Understanding Security Using the OSI Model,” SANS Institute InfoSec Reading Room, Mar. 2002, https://www.sans.org/reading-room/whitepapers/protocols/understanding-security-osi-model-377.

[81] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger, “Measuring small subgroup attacks against Diffie-Hellman,” in *Network and Distributed System Security Symposium (NDSS ’17)*, Feb. 2017.

[82] P. C. van Oorschot and M. J. Wiener, “On Diffie-Hellman key agreement with short exponents,” in *EUROCRYPT*, 1996.

[83] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J. A. Halderman, “Towards a Complete View of the Certificate Ecosystem,” in *Internet Measurement Conference (IMC ’16)*, Nov. 2016.

[84] N. Vratonjic, J. Freudiger, J.-P. Hubaux, and M. Felegyhazi, “Securing Online Advertising,” Tech. Rep., 2008.

[85] D. Wong, “How to backdoor Diffie-Hellman,” Cryptology ePrint Archive, Report 2016/644, 2016, http://eprint.iacr.org/2016/644.

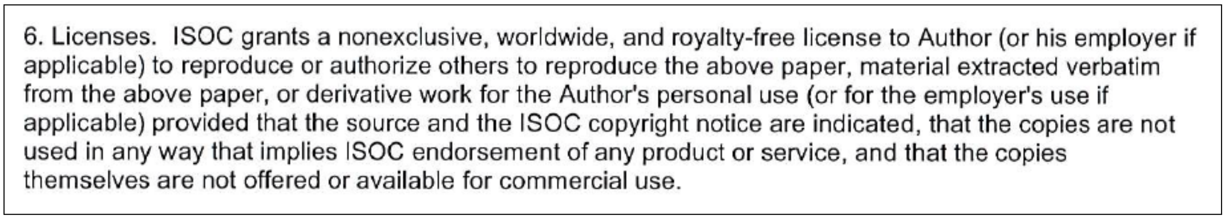
[86] ——, “Socat? What? (timeline of events),” 2016, https://github.com/mimoo/Diffie-Hellman_Backdoor/tree/master/socat_reverse.

[87] T. Ylonen, “The Secure Shell (SSH) Transport Layer Protocol,” Jan. 2006, https://tools.ietf.org/html/rfc4253.

Appendix A

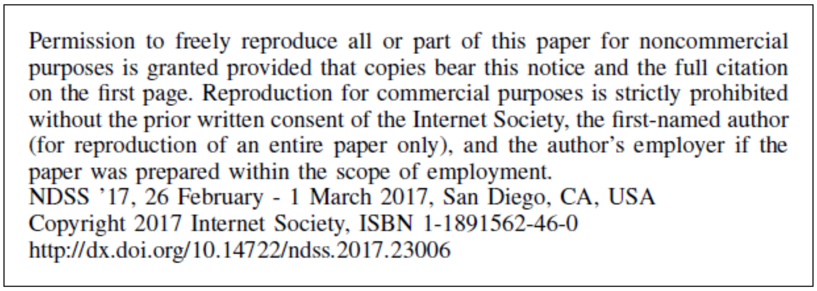
Permission to Reproduce Article Material

Figures A.1 and A.2 allow the author to reproduce material from [35] for this thesis.



6. Licenses. ISOC grants a nonexclusive, worldwide, and royalty-free license to Author (or his employer if applicable) to reproduce or authorize others to reproduce the above paper, material extracted verbatim from the above paper, or derivative work for the Author's personal use (or for the employer's use if applicable) provided that the source and the ISOC copyright notice are indicated, that the copies are not used in any way that implies ISOC endorsement of any product or service, and that the copies themselves are not offered or available for commercial use.

Figure A.1: **License from ISOC.** The License section of the copyright form filled out for [35] provides the author license to reproduce material from the paper.



Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23006

Figure A.2: **Permission Notice.** The permission notice displayed on the first page of [35] provides the author license to reproduce material from the paper if this notice is displayed.

Appendix B

Companies Found in Connection to Name Mismatch Errors

The full list of companies found in connection to name mismatch errors is provided here. Each is specified by a keyword based on the company's website or based on its FQDN known to be used on certificates.

'akamai', 'cloudfront', 'cachefly', 'cdnetworks', 'chinacache', 'cloudflare', 'CloudFlare', 'distilnetworks', 'edgecastcdn', 'fastly', 'googleusercontent.com', 'appspotpreview.com', '.hpe.', 'incapsula', 'instartlogic', 'leaseweb', 'limelight', '.ovh.', 'xserver.jp', 'wpx.jp', 'xtwo.ne.jp', 'fc2.com', 'github', 'godaddy', 'secureserver.net', 'sakura.ne.jp', 'hostmonster', 'netowl', 'axspace', 'secure.ne.jp', 'easyhebergement', 'sedoparking', 'herokuapp', 'home.pl', 'ipage', 'webhostbox.net', 'heteml', 'hostgator', 'websitewelcome.com', 'webfaction', 'dinaserver', 'chinanetcenter', 'hoster.kz', 'speedhost247', 'freehost.com.ua', 'arvixe', 'valuehost.ru', 'reklam9', 'chaturbate', 'hekko.pl', '.reg.ru', 'bigrock', 'yahoo.com', 'secure.hostingprod.com', 'ucoz.net', 'ucoz.ru', 'sharpschool.com', 'tumblr.com', 'notarius', 'hc.ru', 'securedata.net', 'webempresa', 'fozzyhost', 'mchost.ru', 'gridserver.com', 'bizland', 'bluehost.com', 'forumotion', 'inmotionhosting', 'kasserver.com', 'mylittledatacenter.com', 'rozblog.com', 'gudzonhost.ru', 'gmoserver.jp', 'fornex', 'wildfanny.com', 'webhosting.com', 'registrarservers.com', 'tistory', 'webhost1.ru', 'nyi.net', 'nexcess.net', 'dp.tb.ask.com', 'justhost.com', 'jino.ru', 'godo.co.kr', 'sixcore', 'snakeoil.dom', 'trafficplanethosting.com', 'wordpress', 'wpengine.com', 'strikingly.com', 'myinsales.ru', 'accountservergroup.com', 'webserversystems.com', 'lunarpages', 'cyon.ch', 'townsquaremedia', 'acquia', '4hu.com', 'pointhq.com', 'mediacenter.hu', 'valuedomain', 'top10bestvpn', 'asoshared.com', 'azure', 'yourserver.de', 'notexist.com', 'wedos.ws', 'sdska.ru', 'rugion.ru', 'myqcloud.com', 'allinternet.jp', 'sony.', 'sonypictures', 'synology.com', 'timeweb', 'alynx', 'ning.com', 'unoeuro', 'artfiles.de', 'webshopapp.com', 'sucuri', 'firstfind.nl', '123secure.com', 'bravehost.com', 'mapf.com', '163.com', 'rackset.com', 'securesecure.co.uk', 'netangels.ru', 'hostland.ru', 'sidearmsports.com', 'nfadmin.net', 'tarhely.eu', 'cafe24', 'arvancloud', 'snjtoday.com', 'vozpopuli.com', 'andar.co.kr', 'trsprtr2.com', 'websiteseguro.com', 'weebly.com', 'sgvps.net', 'parseek.com', 'gridhost.co.uk', 'hostinger.com', 'hostingplatform.com', 'nazwa.pl', 'linuxpl.com', 'srv.cat', 'infomaniak', 'xrea.com', 'squarespace.com', 'opentransfer.com', 'myserverhosts.com', 'zenbox.pl', '*.*', 'makeshop.jp', 'ehosts.com', 'businesscatalyst.com', 'websitehostserver.net', 'agava.net', 'turhost.com', 'mirtesen.ru', 'alfahostingserver.de', 'mybigcommerce.com', 'bizmw.com', 'maintenis.com', 'eurobyte.ru', 'blog.me', 'kinghost.net', 'elsevierhealth.com', 'ferozo.com', 'valueserver.jp', 'serveriai.lt', 'lineapps.com', 'sslblindado.com', 'vpsprivate.net', 'hoster.by', 'myregisteredsite.com', 'loopiasecure.com', 'webhostinghub.com', 'ioservers.com', 'publigo.fr', 'newscyclecloud.com', 'vshosting.cz', 'aruba.it', 'tmall.com', 'myshopify.com', 'livejournal.com',

’pantheonsite.io’, ’blog.ir’, ’jimdo.com’, ’civicplus.com’, ’schoolwires.net’, ’m3xs.net’, ’justsize’, ’webspaceverkauf.de’, ’krystal.co.uk’, ’venez.fr’, ’ktnet.kg’, ’planethoster’, ’aliyuncs.com’, ’kalalist.com’, ’speedweb.sk’, ’hostoffshore.com’, ’proginter’, ’.tom.com’, ’naltis’, ’cdn77.com’

Curriculum Vitae

Name:	Kristen Dorey
Post-Secondary Education and Degrees:	Western University London, Ontario 2014: B.E.Sc. (Chemical Engineering)
Honours and Awards:	2015: NSERC Canada Graduate Scholarship (CGSM) 2015: Ontario Graduate Scholarship 2016: Ontario Graduate Scholarship 2017: Graduate Student Award for Excellence in Research

Publications:

Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. “Indiscreet Logs: Diffie-Hellman Backdoors in TLS,” in *Network and Distributed System Security Symposium (NDSS ’17)*, Feb. 2017.