On the Design and Implementation of Secure Network Protocols

Nadhem J. AlFardan

Thesis submitted to

Royal Holloway, University of London

for the degree of

Doctor of Philosophy

2014

Declaration

I, Nadhem J. AlFardan, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed: ....

(Nadhem J. AlFardan)

Date: ....

"Two roads diverged in a wood, and I took the one less traveled by, and that has made all the difference."
Robert Frost

This doctoral thesis is lovingly dedicated to: My parents, Jawad and Alawia, may their souls rest in peace. To my sisters, Zoya and Zahra. To my brothers, Mahmoud, Monther and Hussain. There is no doubt in my mind that without their prayers, support and encouragement I could not have completed this journey.

Acknowledgement

First and foremost, I would like to acknowledge my PhD supervisor, Prof. Kenny Paterson, who made this journey worth taking. Throughout my research period, he provided me with encouragement, sound advice, good teaching, and lots of good ideas. His enthusiasm and engagement in my area of research kept me motivated in every stage of my PhD journey. This thesis would not have been possible without his help, support and patience. Thank you.

I wish to thank my extended family, especially Firas, Ghaida and Hassan, who have been supporting me and believing in me all the way.

I would like to thank my friends, especially Salman and Ahmad, who have been the best of friends to me over the years.

Special thanks to David McGrew, David Williamson, Eric Vyncke and my other colleagues at Cisco, for their support and encouragement throughout this experience.

I would like to acknowledge the ISG IT support staff, especially Jon Hart, for providing all what I needed to carry out my laboratory work.

Finally, thanks to everyone who kept asking me "are you done yet?".

Abstract

Network Protocols are critical to the operation of the Internet and hence the security of these protocols is paramount. Our work covers the security of three widely deployed protocols: Domain Name System (DNS), Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Our work shows that the design or implementation of some variants of these protocols are vulnerable to attacks that compromise their fundamental security features. In all of the cases we include experimental results demonstrating the feasibility of our attacks in realistic network environments. We propose a number of countermeasures for the attacks, some of which have already been implemented in practice.

We start by describing the structure of DNS and present a number of existing DNS security protocols. We then focus on DepenDNS, a security protocol that is intended to protect DNS clients against cache poisoning attacks. We demonstrate that DepenDNS suffers from operational deficiencies, and is vulnerable to cache poisoning and denial of service attacks.

We then give an overview of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS), and draw the similarities and differences between the two protocols. We describe the padding oracle concept and present a number of recent attacks against TLS.

We then present new techniques to conduct a full plaintext recovery attack against the OpenSSL implementation of DTLS, and a partial plaintext recovery attack against the GnuTLS implementation of TLS and DTLS. Our attacks exploit timing-based side channels that would not have been exploitable without our new techniques. We also describe countermeasures for the attacks.

We then present new distinguishing and plaintext recovery attacks against all versions of TLS and DTLS and in almost all implementations of the two protocols. Our attacks are based on timing-based side channels and exploit TLS and DTLS design and implementation decisions. We describe how to conduct a full plaintext recovery attack against implementations that follow the standard, and a partial plaintext recovery attack against implementations that do not. We discuss a number of countermeasures for the attacks, and describe their practicality and effectiveness.

We conclude the thesis by discussing the wider implications of our work on the design and implementation of secure network protocols.

Contents

List of Figures

List of Tables

List of Algorithms

List of Notations

We list here notations and symbols that will be consistently used throughout this thesis. All other symbols or notations will be introduced when they are first used.

π	The algorithm used by DepenDNS to accept or reject an IP address.
G_i	The grade of an IP address calculated by the DepenDNS algorithm.
P	A Plaintext message.
R	A Plaintext record.
T	The MAC tag.
b	The block-size of the selected block cipher in bytes.
C_i	A ciphertext block i.
P_i	A plaintext block i.
$B[i]$	Byte at position i in block B.
D	A decryption algorithm of a block cipher.
E	An encryption algorithm of a block cipher.
K_d	The block cipher decryption key.
K_e	The block cipher encryption key.
`HDR`	The (D)TLS header.
`SQN`	The (D)TLS sequence number field.

List of Abbreviations

We list here abbreviations that will be used throughout this thesis. All other abbreviations will be introduced when they are first used.

AE	Authenticated Encryption
AES	Advanced Encryption Standard
ANS	Authoritative Name Server
CBC	Cipher Block Chaining
CDN	Content Delivery Network
DES	Data Encryption Standard
DNS	Domain Name System
DNSSEC	DNS Security Extensions
DOS	Denial of Service
DTLS	Datagram Transport Layer Security
IP	Internet Protocol
MITM	Man-In-The-Middle
RDNS	Recursive Domain Name System
RFC	Request for Comment
RR	Resource Record
RTP	Real Time Protocol
SSL	Secure Socket Layer
TCP	Transport Control Protocol
TLD	Top-Level Domains
TLS	Transport Layer Security
TXID	Transaction Identifier
UDP	Unreliable Datagram Protocol

Chapter 1

Introduction

1.1 Context

The evolution of secure network protocols has been largely driven by the discovery and the successful exploitation of weaknesses in either the design or the implementation of these protocols. The Domain Name System (DNS) and Transport Layer Security (TLS) provide good examples that demonstrate this *broken* reactive model of evolution.

Maintaining the security of DNS has been a continues challenge with high-profile and high-impact attacks frequently emerging (for example, Kaminsky's cache poisoning attack against DNS), which are usually followed by the development of *ad hoc* security protocols that try to protect DNS from these attacks. In most cases, attacks against DNS have exploited trivial, and on occasion known, weaknesses. Most of these attacks would have been prevented if the basic DNS security mechanisms had been deployed.

TLS, on the other hand, is by far the most widely deployed secure network protocol today, and which best show-cases the failure of this *ad hoc* approach of designing and implementing secure network protocols. Attacks of different severity and practicality levels have been published against TLS (and its predecessor, Secure Socket Layer), triggering *ad hoc* and non-coordinated responses from the Internet Engineering Task Force (IETF) who maintain the protocol specification, and the TLS open and closed source code development community. Alarmingly, the number of attacks against TLS has recently been on the rise including, for example, BEAST, CRIME, Lucky 13 (our attack discussed in Chapter 5) and BREACH.

Our work takes advantage of *previously unknown* weaknesses introduced by this *ad hoc* approach to develop attacks that exploit the above mentioned protocols using basic, but novel, techniques.

1.2 Contributions

The work presented in this thesis reflects our analysis of three secure network protocols: DependDNS (a DNS security protocol), TLS and Datagram TLS (DTLS). We describe a number of weaknesses in the design and implementation of these protocols that could be exploited by an adversary to conduct various, and in many cases severe, attacks that undermine their fundamental security goals. The impact and applicability of our work goes beyond academia and extends to cover the larger community of secure network protocol researchers, designers and software implementers. Our work demonstrates *again* the (in)security of the MAC-then-Encode-then-Encrypt (MEE) construction for TLS and DTLS, using new techniques to exploit design and implementation decisions made for the two protocols to build attacks. We describe how to use these new techniques to recover TLS and DTLS-protected plaintext. In addition, we demonstrate that a number of DNS security protocol implementations are impractical and, in the case of DepenDNS, are ineffective.

We took the route of: identifying and verifying potential design or implementation weaknesses, practically exploiting these weaknesses, and then responsibly disclosing our research results. We responsibly disclosed our findings, working closely with the (D)TLS standards' design and development community to address the newly discovered weaknesses; a large number of open source software developers and vendors had to modify their code in response to our attacks. During our research, we collaborated with the authors of the IETF (D)TLS standards, various vendors such as Google and Microsoft, and open source software developers maintaining cryptographic libraries such as OpenSSL and GnuTLS.

Our work promotes further the use of secure TLS modes of operation such as authenticated encryption (AE) and the proper implementation of secure network protocols, while considering clarity, effectiveness, practicality and ease of deployment.

Secure network protocols are implemented as part of a system. The interaction between the secure network protocols and the other components of the system, especially their upper and lower-layers, plays a critical role in defining the system's overall security. We demonstrate this in the different chapters of this thesis. For example, we demonstrate how to use application layer messages to construct a new realisation of Vaudenay's padding oracle [102] in the context of DTLS in Chapter 4.

1.3 Publications

This thesis contains published research materials with K.G. Paterson:

An Analysis of DepenDNS [7]**:** Published in the 13th Information Security Conference (ISC), and forming the basis of Chapter 2.

Plaintext-Recovery Attacks Against Datagram TLS [5]**:** Published in the 19th Network and Distributed System Security (NDSS) Symposium, where we received one of the two distinguished paper awards[1]. The work in [5] forms the basis of Chapter 4.

Lucky Thirteen: Breaking the TLS and DTLS Record Protocols [6]**:** Published in the 34th IEEE Security and Privacy (IEEE S&P) Symposium, and forming the basis of Chapter 5.

1.4 Thesis Outline

In Chapter 2, we give an introduction to DNS, the services it provides and the standard security mechanisms available to protect DNS from denial of service and cache poisoning attacks. We then describe a number of security protocols that have been proposed to further secure DNS and give an overview of their current deployment status. We analyse a particular protocol, DepenDNS, in detail and demonstrate that the protocol is vulnerable to a number of attacks despite the protocol designers' claims. We also describe a number of issues that make DepenDNS impractical to deploy. We conclude the chapter by summarising our findings and giving our perspective on the ongoing efforts to secure DNS.

In Chapter 3, we provide the necessary background information and prerequisite material that are needed to establish an understanding of the TLS and DTLS protocols. We provide background information about the TCP/IP protocol suite and describe three fundamental networking protocols: IP, TCP and UDP. We then introduce Transport Layer Security (TLS), describe how the protocol is structured and discuss its modes of operation. We also introduce Datagram Transport Layer Security (DTLS) and describe the differences between TLS and DTLS. We then present in detail the padding oracle concept and describe how an attack can be theoretically mounted against TLS using the oracle. Finally, we present a number of recent attacks against the two protocols, serving as a forerunner to our attacks on DTLS and TLS, which we present in Chapters 4 and 5.

In Chapter 4, we present our attacks against the OpenSSL implementation of DTLS with the MAC-then-Encode-then-Encrypt construction. We report our experimental results demonstrating efficient and reliable recovery of full DTLS plaintexts in the OpenSSL case. We then discusses how similar attacks can recover partial plaintexts in

[1]`http://www.internetsociety.org/events/ndss-symposium-2012/papers`

the GnuTLS implementation of TLS and DTLS.

In Chapter 5, we present a family of attacks that apply to the MAC-then-Encode-then-Encrypt construction in all TLS and DTLS implementations that are compliant with TLS 1.1 or 1.2, or with DTLS 1.0 or 1.2. Our attacks also apply to implementations of SSL 3.0 and TLS 1.0 that incorporate padding oracle attack countermeasures. We first provide further background on the HMAC calculation. We then present the basic distinguishing attack against RFC-compliant implementations of TLS and DTLS, followed by a description of our plaintext recovery attacks in the context of TLS. We explain how to modify them to apply to DTLS. We report on the experimental validation of our attacks for the OpenSSL implementation. We then describe the modifications needed to make our attacks applicable to other implementations. We finally give guidance on how to implement the MAC-then-Encode-then-Encrypt construction so as to avoid the attacks.

We conclude the thesis by discussing the wider implications of our work on the design and implementation of secure network protocols.

Chapter 2

The Domain Name System

2.1 Introduction

We start the chapter by giving an introduction to the Domain Name System (DNS), the services it provides and the standard security mechanisms available to secure DNS against denial of service and cache poisoning attacks. We then describe a number of security protocols that have been proposed to further secure DNS and give an overview of their current deployment status. We analyse a particular protocol, DepenDNS, in detail and show that the protocol is vulnerable to a number of attacks despite the protocol designers' claims. We also describe a number of issues that make DepenDNS impractical to deploy. We conclude the chapter by summarising our findings and giving our perspective on the ongoing efforts to secure DNS.

2.2 Introduction to the Domain Name System

The Domain Name System (DNS) [65, 66] is a fundamental service that is critical to the proper operation of the Internet. While people are quite good at remembering names, they are not good at remembering IP addresses and hence the need for DNS to help translate names to IP addresses that the Internet can route. The most common service that DNS provides is mapping names to IP addresses (for example, translating "www.example.com" to 192.0.43.10). In addition to mapping names to IP addresses, DNS provides other services such as mapping IP addresses to names (commonly referred to as reverse DNS) and assigning aliases to domain names.

Domain names, IP addresses and other information are maintained on DNS servers in the form of persistent or cached entries referred to as Resource Records (RRs). RRs share the structure described in RFC 1035 [66] and shown in Figure 2.1.

```
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                     NAME                      /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     TYPE                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    CLASS                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     TTL                       |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   RDLENGTH                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
/                    RDATA                      /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2.1: Resource Record structure. The structure shows the fields in a RR that are explained in Section 2.2.

The 2-byte TYPE field shown in Figure 2.1 contains the RR type code. Examples of commonly used types include:

- `A`: The address record used for serving IPv4 host addresses.
- `AAAA`: The address record for serving IPv6 host addresses.
- `CNAME`: The canonical name record used for serving an alias of a host name. Multiple `CNAME`s can be created such that the same host name that a user queries would resolve to multiple canonical names pointing to different IP addresses, hosted on different servers that could be geographically distributed. Figure 2.2 shows an example where two `CNAME`s are created to serve "www.gov.uk"[1].
- `MX`: The mail exchange record used for serving the mail server host name.
- `NS`: The name server record used to identify the authoritative name server hosting the domain.
- `PTR`: The pointer record used in reverse DNS look-ups to map an IP address to a host name.

[1]The reader may receive a different output when trying the same DNS query. The values he receives would largely depend on the source of the DNS requests.

```
www.gov.uk canonical name = www.gov.uk.edgekey.net.
www.gov.uk.edgekey.net canonical name = e6452.b.akamaiedge.net.
Name: e6452.b.akamaiedge.net
Address: 23.14.4.23
```

Figure 2.2: DNS look-up showing two `CNAME` RRs being served for the host name "www.gov.uk".

The time-to-live (TTL) value associated with an RR specifies how long, in seconds, should DNS clients or resolvers cache an RR. For example, a TTL value of 3600 indicates that the RR should be cached for only an hour, while a TTL value of 0 indicates that the RR should not be cached. The receiver of an RR can choose to ignore the TTL value it receives and assign a value of its choice.

The other fields that make up an RR include:

- NAME: The domain name that owns the RR.
- CLASS: The CLASS code. The reader may think of this field as the category of the record and is typically set to 1 for an Internet CLASS.
- RDLENGTH: The length of the RDATA field in octets (bytes).
- RDATA: A variable length string that describes the RR. The format of this record varies according to the TYPE and CLASS of the RR. For example, the RDATA field may contain a host name and an IPv4 address in case it was for an `A` RR.

DNS can be thought of as a distributed database with a hierarchical structure that is made up of name servers hosting the database and serving RRs. This hierarchy is shown in Figure 2.3. The root domain (".") is at the top of the hierarchy and is served presently by thirteen root operators with servers distributed around the world[2]. The list of the current root servers is shown in Table 2.1. Typically, DNS server packages (for example, ISC BIND[3] and Microsoft DNS[4]) would embed this information in their code. This clearly makes changing the IP address of a root server a daunting task. The second level in the hierarchy contains top-level domains (TLDs) that can be classified as generic (gTLD) such as ".com", or country code (ccTLD) such as ".uk". Multiple levels exist underneath the TLDs. The domain names located in the lower levels are generally served by their corresponding organisations. For example, the Internet Assigned Numbers Authority (IANA)[5] hosts the domain "example.com" [3].

[2]http://www.root-servers.org
[3]http://www.isc.org/downloads/BIND
[4]http://technet.microsoft.com/en-US/network/bb629410.asp
[5]http://www.iana.org

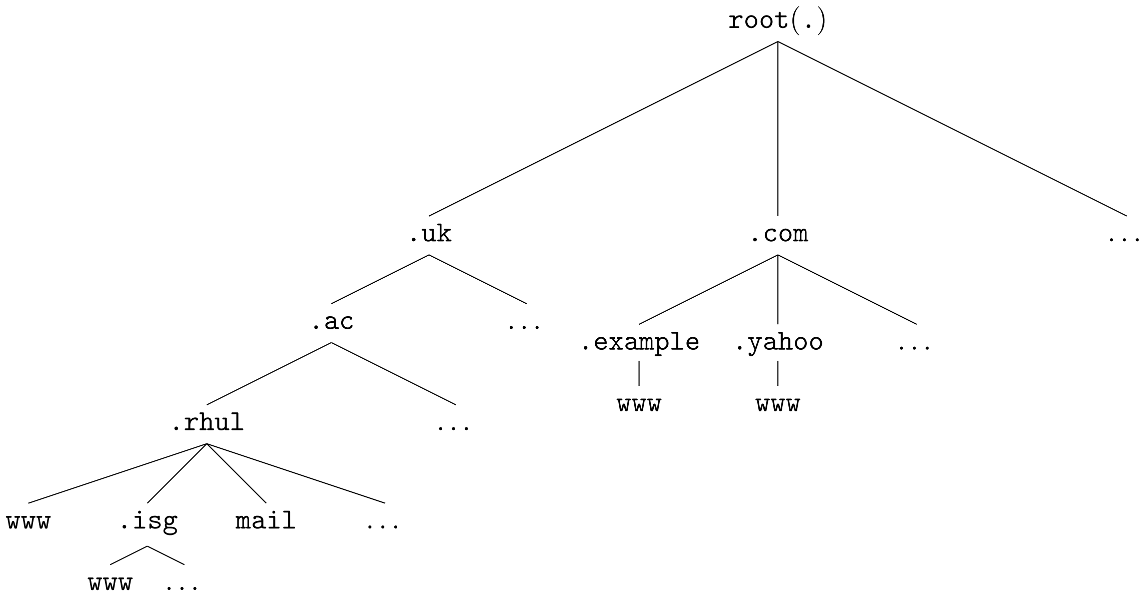


Figure 2.3: The Domain Name System structure. The figure shows the DNS root domain (".") in the top of the hierarchy. The figure also show two TLDs, ".com" and ".uk" and sample levels that exist under the ".uk" TLD.

The operation of DNS is based on queries (requests) and responses (replies). A client initiates the process by sending a DNS resolution query for a host name (for example, "www.example.com") to its DNS resolver which in return searches its cache entries for the name being requested. If an entry does not exist, resolvers may go through a *recursive* DNS look-up process that starts from the root servers and continues all the way down to the authoritative name servers (ANSs) responsible for hosting the domain being requested (for example "example.com"). Resolvers capable of performing this recursive DNS look-up are referred to as recursive DNS (RDNS) resolvers; we refer to them in this chapter as resolvers in short. A resolver can also be configured to forward DNS queries to other resolvers (referred to as upstream resolvers) to perform recursive DNS look-ups on its behalf.

Information about domains and their records are contained within DNS zones. An ANS maintains the zone's database and responds to DNS queries for hosts that exist in the zone. Upon receiving an answer from an ANS, a resolver caches and forwards the answer to the requesting client.

2.2.1 DNS and Content Delivery Networks

Content Delivery Networks (CDNs) are built to enhance the user's experience when trying to access an Internet resource like a website. A Content Delivery Network

Root Server Host Name	IP Address	Operator
a.root-servers.net	198.41.0.4	VeriSign, Inc.
b.root-servers.net	192.228.79.201	University of Southern California (ISI)
c.root-servers.net	192.33.4.12	Cogent Communications
d.root-servers.net	199.7.91.13	University of Maryland
e.root-servers.net	192.203.230.10	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4	US Department of Defence (NIC)
h.root-servers.net	128.63.2.53	US Army (Research Lab)
i.root-servers.net	192.36.148.17	Netnod
j.root-servers.net	192.58.128.30	VeriSign, Inc.
k.root-servers.net	193.0.14.129	RIPE NCC
l.root-servers.net	199.7.83.42	ICANN
m.root-servers.net	202.12.27.33	WIDE Project

Table 2.1: DNS root servers.

consists of a set of surrogate servers distributed around the world. The surrogate servers are deployed in multiple locations in order to optimise the end user experience by choosing the nearest surrogate server to the user [101]. For example, web requests generated by a UK-based end user for a website hosted by a CDN will generally be served by a surrogate server that is located in the UK. Most CDN providers deploy DNS redirection to forward the client's request to the closest server containing the resource being requested. One of the characteristics of DNS records served by CDNs is that they have a low TTL value. Serving a low TTL value causes more frequent DNS look-ups for the same host name, allowing the DNS server hosting the DNS entry to possibly serve different IP addresses, based on factors such as the availability of the surrogate servers or the client's proximity to the surrogate servers. By way of example, the following shows the TTL value for "134.g.akamai.net", which is the `CNAME` RR corresponding to "www.live.com". It can be seen that the TTL value is set to only 20 seconds.

```
$ dig www.live.com
...
www.live.com. 1216 IN CNAME search.msn.com.edgesuite.net.
search.msn.com.edgesuite.net. 2382 IN CNAME a134.g.akamai.net.
a134.g.akamai.net. 20 IN A 88.221.94.72
a134.g.akamai.net. 20 IN A 88.221.94.34
...
```

CDNs have proven lately to be a very attractive option for hosting rich web content such as video. In fact, high profile websites such as YouTube, CNN and BBC commonly make use of commercial CDNs such as Akamai and Limelight [96].

2.3 Security of DNS

Threats targeting DNS and DNS caches in particular are not new. They have existed since the day DNS was introduced. However, the topic gained significant visibility and attention after a number of high-profile attacks such as Kaminsky's DNS cache poisoning attack [50], described in Section 2.3.3. Kaminsky discovered a fundamental flaw within DNS implementations that could allow remote attackers to corrupt the cache of a DNS server within a matter of seconds, exploiting a combination of old (known) and new (previously unknown) vulnerabilities.

Attacks against DNS can be classified as denial of service (DoS) or cache poisoning attacks. The former targets the availability of the DNS service or data, while the latter targets the integrity of the DNS data. Typically, denial of service attacks target ANSs while cache poisoning attacks target resolvers. DNS information served over the Internet is considered public. Guaranteeing the integrity and authenticity of the Internet DNS data are of top concern. Although confidentiality is generally not a concern, keeping DNS information confidential might be required in private networks (for example, networks serving internal corporate users). A good description of threats against DNS is given in RFC 3833 [13].

2.3.1 DNS Cache Poisoning

Cache poisoning, in the context of DNS, refers to act of *intentionally* corrupting the data contained in DNS caches. DNS messages, including queries and responses, are communicated in clear using User Datagram Protocol (UDP) [77], and possibly Transmission Control Protocol (TCP) [79, 18], with no integrity check mechanisms in place [66], other than the basic, non-cryptographically generated, UDP and TCP checksums that are mainly targeted to detect network errors. We provide further background information on UDP and TCP in Chapter 3 of this thesis. The lack of a proper integrity check mechanism makes DNS vulnerable to attacks involving unauthorised data modification, in which an attacker may alter the data in various ways, with an ultimate objective of poisoning the content of a resolver's cache, or possibly a DNS client's cache. Such attacks are referred to as DNS cache poisoning and they present potential security threats to users. For example, a user can unknowing be redirected to a malicious web site, which mimics the actual one in attempt to harvest the user's personal information;

all as a result of receiving a poisoned DNS RR pointing to the malicious web server IP address.

DNS cache poisoning is commonly achieved through *blind* injection of spoofed DNS replies, where the attacker has no access to the information contained in the original DNS query. This becomes a trivial attack in the man-in-the-middle (MITM) setting, in which the attacker has visibility of all DNS data and hence can respond to DNS queries using information of his choice.

In a cache poisoning attack, the attacker may target:

- Resolvers: These servers contain cached entries and generally serve a large number of users. An attacker may try to poison the resolver's cache for a domain or a number of domains, or even try to gain control of the underlying system running the DNS service.

- Clients: Although spoofing DNS responses from a resolver to a particular DNS client is possible, this might not be a cost effective attack, unless the client under attack is of high-value to the attacker.

The attacker can also target ANSs. These servers contain persistent DNS entries for zones. An attacker may try to control a zone (for example, ".rhul.ac.uk") by controlling the underlying system running the DNS service. An attacker gaining control of a server acting as an ANS would have full control over the DNS zone. Attackers taking control of ANSs that are high in the DNS hierarchy (for example ".com") can cause major Internet service disruption. However, an attack of this type does not fall under the cache poisoning category. In the next sections, we further describe the cache poisoning attack in the context of resolvers.

To successfully poison the cache of a DNS server, an attacker may spoof a DNS response and deliver it to the requester ahead of the legitimate one. Subsequent responses for the same DNS query should be ignored by the requester as per RFC 1034 [65]. To be accepted by the requester, the spoofed response must also pass the standard security controls incorporated within DNS. These include comparing the DNS 16-bit transaction identifier (TXID) and the randomised UDP source port in queries and responses [46]. The value of TXID is assigned by the program running the DNS service while the UDP source port assignment is handled by the underlying operating system. In RFC 6335 [27], IANA divides port numbers into three ranges:

- 0 - 1023: Well known ports;

- 1024 - 49151: Registered ports;

- 49152 - 65535: Ephemeral ports.

Operating systems are expected to use the ephemeral port range when assigning random UDP source ports for DNS queries. However, operating system developers can choose to ignore this and use a smaller pool of ports for UDP source assignment or even assign sequential UDP source ports. For example, Windows Server 2008 assigns a random source UDP port numbered 49152 or above[6], while earlier versions of Microsoft Windows assign it from a much smaller range of ports[7], only 1024 to 5000. The Linux 2.4 kernel is configured by default to assign a port from the range 32768 to 61000. This clearly shows that DNS heavily relies on the specific implementation of the underlying operating system for the UDP source port assignment. Operating systems not properly randomising the UDP source port was one of the weaknesses that Kaminsky exploited in his attack [50].

Using a *random* TXID value introduces 16 bits of entropy, while using a *random* UDP source port adds a variable amount of entropy that depends on the operating system configuration and the number of already used UDP ports. This added entropy makes it harder to perform blind injection.

The following are definitions for some of DNS related concepts that we will use in the coming sections.

Definition 2.1. Window of opportunity: The moment right after sending a DNS query to the moment right before the arrival of the (first) valid response. This applies to queries and responses between a resolver and an ANS or between a resolver and its upstream resolver, if it was configured with one. This also applies to queries and responses between a client and its resolver. This definition assumes that the domain name being requested is not in the cache of the resolver or the client. Responses received outside the window of opportunity should be rejected by the DNS query initiator [66].

Definition 2.2. Outstanding DNS query: A DNS query that has been sent by a resolver or a client and is waiting for a response.

Definition 2.3. Valid DNS Response: A DNS response that meets the security requirements enforced by the initiator of the DNS query.

For example, the security requirements could be matching the TXID and the UDP source port contained in the original query.

Definition 2.4. Collision: Having two or more valid DNS responses for the same query.

[6] `http://technet.microsoft.com/en-us/library/dd197515(v=ws.10).aspx`
[7] `http://support.microsoft.com/kb/832017`

In the case of a collision, typically, one of the responses will be spoofed by the attacker and the other will be the legitimate response.

Definition 2.5. DNS blind injection: An injection of a DNS response by an attacker with the goal of causing a collision and ultimately poisoning the DNS cache.

There are cases, as we see later, in which the attacker has partial information about the legitimate DNS response (for example, the domain name being requested). We refer to this type of injection as *partially sighted* injection.

RFC 5452 [46] gives an overview of the success probability of DNS cache poisoning attacks. The success probability, P, of an attacker poisoning the cache of a resolver after n spoofed DNS responses that arrive *within* the window of opportunity is equal to n divided by the size of the DNS problem space, i.e. the number of possible TXID and UDP port combinations. This assumes that the query is for a host name that is not in the resolver's cache. The value of P is calculated as:

$$P = \frac{n}{N \cdot U \cdot T}, \tag{2.1}$$

where N is the number of ANSs serving the domain name being requested, with an average value of around 2.4 according to [46], U is the number of available UDP ports to choose from, and T is the number of TXIDs available (maximum of 2^{16}). Here, we assume that the TXID and UDP source port are randomly selected.

The attacker can initiate queries for domain names that are *not* in the cache of a resolver and follow that with spoofed responses in a partially sighted injection mode, in which the attacker has information about the domain name being requested. We would expect that the attacker can increase the number of concurrent queries for the same domain to increase the success probability. If this is the case, then for d simultaneous outstanding queries, the success probability, P_d, is calculated as:

$$P_d = \frac{n \cdot d}{N \cdot U \cdot T}. \tag{2.2}$$

Equation (2.2) applies when the value of d is small, relative to the amount of entropy available. As the value of d increases, the attacker can start exploiting the "birthday paradox"[8], significantly raising the chance of success. Therefore, for d incoming queries to the same domain name, DNS resolvers are expected to rate-limit the number of requests they send to the same ANS hosting the domain name being queried [46]. This is to thwart the birthday paradox attack, which we further discuss in Section 2.3.2.

If we assume that the window of opportunity is W seconds in size and that the

[8]http://www.secureworks.com/resources/articles/other_articles/dns-cache-poisoning

attacker can send r spoofed responses per second, all arriving within the window of opportunity, then the success probability can be calculated as:

$$P = \frac{W \cdot r}{N \cdot U \cdot T}, \tag{2.3}$$

where we assume $d = 1$.

Some resolver implementations contained flaws in the TXID or the source UDP port randomisation processes, resulting in an insufficient amount of entropy in DNS queries and making it easier for an attacker to predict the values of the TXID or the source UDP port [50]. Other situations include the implementation of Network Address Translation (NAT) in front of DNS resolvers. A NAT device may replace the original random source port with a sequential one of its choice, also causing a reduction in the unpredictability of the UDP source port. We assume that the attacker has no access to the DNS query and hence must guess the two random variables, the TXID and the UDP source port. A more severe scenario is when the attacker acts as a MITM. In this case, attackers have visibility of all DNS data and hence can respond to DNS queries using information of their choice.

2.3.2 DNS Cache Poisoning and the Birthday Paradox

Some implementations of DNS such as old versions of ISC BIND (version 9.2.8 and earlier[9]) send simultaneous DNS requests to the same ANS for the same domain name. An attacker can take advantage of this behaviour by sending d DNS queries followed by an equal or higher number of spoofed DNS responses, arriving within the window of opportunity, to exploit the "birthday paradox". The attack was first published in 2002[10] and gives the attacker an opportunity to match a valid response using fewer spoofed DNS responses, hoping for a collision. The more DNS queries the attacker sends, accompanied with an equal or higher number of spoofed responses that arrive within the window of opportunity, the greater the probability of collision.

If the attacker issues d DNS queries for the same domain name served by the same ANS, and sends d spoofed responses, then the probability of a *collision* (having two or more valid DNS responses for the same query) in the DNS responses which would result in successfully poisoning the DNS cache can be calculated using the following lower bound formula:

$$P = 1 - \left(1 - \frac{1}{U \cdot T}\right)^{d(d-1)/2}. \tag{2.4}$$

[9]http://www.isc.org/downloads/BIND
[10]http://www.kb.cert.org/vuls/id/457875

The attack becomes more serious when an implementation fails to properly randomise the values of the TXID or the UDP source port. To protect against the birthday attack, resolvers should be configured to limit the number of requests they send for a domain name. For example, Google Public DNS[11], a free global DNS resolution service, never allows more than a single outstanding query on one of its resolvers, for the same domain name, query type, and ANS IP address.

To demonstrate the effect of exploiting the birthday paradox, let us take the case when an implementation fails to randomise the UDP source port, leaving TXID as the only source of entropy. Figure 2.4 shows the success probability, calculated using equation (2.4), as the value of the number of outstanding queries, d, increases. The attacker can achieve a success probability of 0.5 with only 300 outstanding queries and matching spoofed responses, and 0.99 with only 776 outstanding queries and matching spoofed replies; compare this to Figure 2.5, when only one query is sent from the resolver to the ANS. In both cases, we assume that the value of the 16-bit TXID has been randomly assigned.

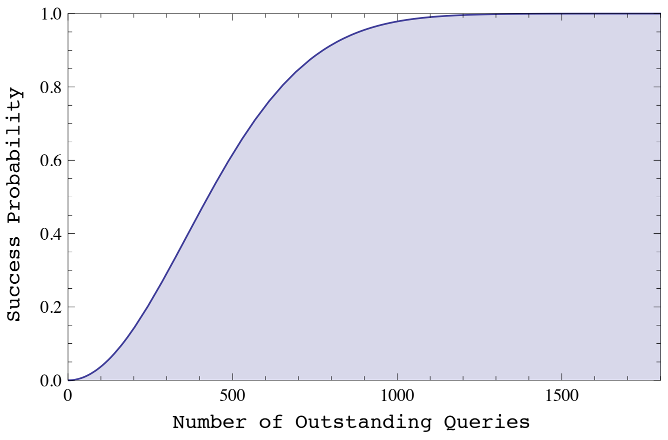


Figure 2.4: Birthday attack success probability with variable d outstanding queries and when relying on a random TXID only. The figure shows that the attacker can achieve a success probability of 0.5 with only 300 outstanding queries and matching spoofed responses, and 0.99 with only 776 outstanding queries and matching spoofed replies

2.3.3 Kaminsky Attack Against DNS

In this section, we give a short overview of Kaminsky's attack [50], one of the most high-profile attacks against DNS. Kaminsky's cache poisoning attack against DNS exploited two basic vulnerabilities in a number of DNS implementations. Not properly randomising the source UDP port is a known weaknesses that Kaminsky exploited in his attack,

[11]https://developers.google.com/speed/public-dns/docs/security.html

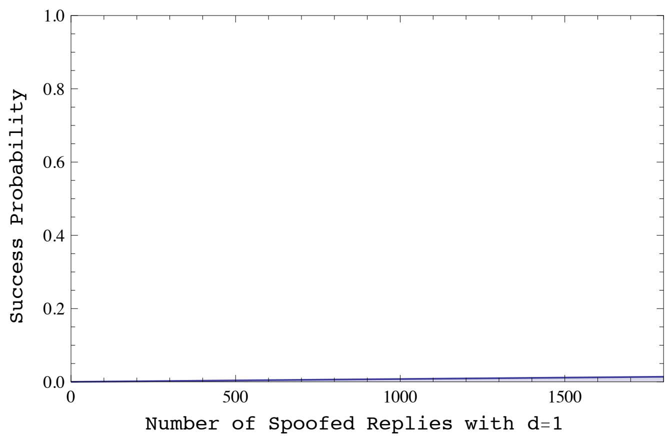


Figure 2.5: Birthday attack success probability with only one outstanding query and when relying on a random TXID only. The reader would compare this to Figure 2.4, which shows the birthday attack success probability with variable d outstanding queries and when relying on a random TXID only

taking advantage of the fact that many implementations have ignored randomising the source UDP port. He also exploited a previously unknown vulnerability that allows an attacker to overwrite a cached DNS RR, even if the RR's TTL has not expired. Let us assume that the attacker tries to poison a resolver's cache `NS` entry for the domain name "foo.com". If successful, then the attacker can serve DNS queries generated by this resolver for any host name under "foo.com" (for example "www.foo.com") using data of his choice. This will impact all the clients that are configured to use the targeted resolver. The attack proceeds as follows.

The attacker sends DNS queries for different random host names under "foo.com", which are unlikely to be in the resolver's cache (for example "123456.foo.com"), forcing the resolver to query the ANS serving "foo.com" for every queried host name. The attacker simultaneously floods the resolver with spoofed DNS responses, but with DNS delegation information that contains a forged `NS` RR pointing to a DNS server controlled by the attacker. A vulnerability existed in many DNS implementations which allowed the overwrite of cached RRs using delegation information contained in DNS responses, and which the resolver will happily accept. Kaminsky demonstrated that combining lack of proper UDP source port randomisation and this vulnerability results in a severe attack against DNS, allowing an attacker to poison a resolver's cache in a very short period of time, in matter of seconds in some cases. The reader can refer to [39] for greater detail on the attack.

2.4 Securing DNS Against Cache Poisoning Attacks

The amount of entropy introduced by using random TXID and random UDP source port has proven to be insufficient to protect against cache poisoning attacks, mostly because of implementation issues[12,13,14,15]. This has resulted in various security protocols being proposed to add further controls to secure DNS from cache poisoning attacks. These protocols do not eliminate the use of a random TXID and a random UDP source port; they build on-top of these basic DNS security controls and are executed only when the standard DNS security checks pass. These protocols use techniques that can be implemented in clients, resolvers, ANSs, or a combination of them. Examples of such protocols include Domain Name Cross Referencing (DoX) [108], `0x20`-Bit encoding [29], ConfiDNS [76], WSEC DNS [75] and DNS Security Extensions (DNSSEC) [1]. In this section, we give an overview for some of these protocols before focusing on DepenDNS [97], the protocol that we analyse in more detail in Section 2.5.

2.4.1 Domain Name Cross Referencing

Domain Name Cross Referencing (DoX) [108] is a resolver-based protocol that forms peer-to-peer networks of resolvers. Resolvers in a DoX peer-to-peer network establish and maintain verification channels. A resolver joins a verification channel and gets assigned k random peers, where a peer is just another participating resolver in the same peer-to-peer verification channel. The exact process of how a resolver joins a verification channel is described in [108].

In DoX, each resolver maintains its own verification cache, vCache, which contains record entries that have been previously verified by DoX. The vCache is kept separate from the standard DNS cache that is maintained by the DNS program running on the system. DNS response messages received from ANSs are evaluated by DoX's consistency check shown in Algorithm 1. A resolver running DoX accepts a DNS response, R^d, if an entry for the domain name being queried exists in its vCache, R^v, and $R^v = R^d$, else the resolver checks if R^d is consistent with what its k peers have in their vCaches. It does this by sending R^v and R^d to its k peers. Every peer is expected to compare R^d with its vCache entry for the domain name in R^d or with a fresh response from the ANS hosting the domain name in question. Every peer would then reply back to the resolver with `Agree`, `Disagree` or `DiffView`:

- An `Agree` reply indicates to the resolver that the peer found R^d to be legitimate,

[12] `http://www.cert.org/advisories/CA-1997-22.html`
[13] `http://www.kb.cert.org/vuls/id/457875`
[14] `http://www.cvedetails.com/cve/CVE-2008-1447`
[15] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-2926`

either by finding a similar entry in its vCache or after contacting the ANS serving the domain name contained in R^d and receiving a response, R^a, that is similar to R^d.

- A `Disagree` reply indicates that the peer found R^d to be different from the version it has in its vCache and different from the response it received from the ANS serving the domain name, R^a.
- A `DiffView` reply indicates that the peer found R^v, which it received from the resolver, to be different from the entry it has in its vCache. However, it found $R^a = R^d$.

A threshold value, *thresh*, is maintained by the resolver and is used to accept or reject R^d. If no peer replies with `Disagree` then the record is accepted by DoX. Else, the resolver issues a fresh query to the ANS serving the domain name being requested. The DNS response is then saved to R^a. R^d is accepted and the resolver's vCache is updated, when $R^a = R^d$ and the number of received `Agree` replies is higher than *thresh*. Else R^d is rejected by the resolver and is assumed to be the result of a cache poisoning attempt.

An issue that DoX must address is how to populate an empty vCache on start-up. The authors of [108] propose a safe start-up phase for resolvers to build up their vCache. However, they do not give details on how to implement this safe start-up phase. We are not aware of practical DNS implementations that have made use of DoX.

2.4.2 `0x20`-Bit Encoding

`0x20`-Bit encoding [29] is another resolver-based protocol that tries to increase the entropy size beyond what the random TXID and the UDP source port provide. It achieves this by encoding DNS queries using a combination of lower and upper case characters, i.e randomising the case pattern in name requests for the ranges (A..Z) and (a..z), (`0x41`..`0x5A`) and (`0x61`..`0x7A`) respectively. The protocol's encoding relies on ANSs retaining the original string in their response and ANSs replying to any case pattern, i.e. it is required that DNS queries are case-*insensitive*. For example, DNS queries for "Www.rHul.AC.uk" and "www.rhul.ac.uk" would resolve to the same IP addresses, while preserving the case pattern in the responses. This behaviour follows the standard, RFC 1034 [65], which states that no significance should be attached to the case of domain names, i.e. two names with the same spelling but different case pattern are to be treated as if identical.

The authors of [29] propose the following simple algorithm to produce an `0x20`-encoded domain name:

Algorithm 1: DoX Consistency Check Algorithm

```
input  : DNS query received from the DNS client, Q
input  : $k$ DoX peers received after joining a verification channel
input  : $thresh$
output: Poison Detected, OK or WARNING

$R^d \leftarrow$ DNS_Lookup(Q);
$R^v \leftarrow$ vCache_Lookup(Q);
if $R^d = R^v$ then
    return OK;
else
    Send $(R^v, R^d)$ to $k$ peers and get the first $m$ results;
    If #Disagree = 0 return OK;
    else
        $R^a \leftarrow$ ANS_Lookup(Q);
        if $R^a \neq R^d$ then
            Poison Detected;
        else if $R^a = R^d$ and #Agree $> thresh$ then
            return OK;
        else
            return WARNING;
```

1. The resolver normalises the DNS query, or response, field by converting each letter in the domain name to lower case.
2. The resolver encrypts the normalised version using some algorithm such as AES.
3. The resolver uses the output of Step 2 to encode the domain name such that:
   (a) if the ith bit of the encrypted output is 0, then make the ith letter in the queried domain name upper case, i.e. perform an OR operation between the character and `0x20`, hence the name of the protocol.
   (b) else make the ith letter in the queried domain name lower case.

The resolver then sends the encoded version of the query to the ANS hosting the domain name or to its upstream resolver if it was configured with one. Upon receiving the DNS response, the resolver applies the same algorithm to the DNS response and compares the newly produced encoded version with the name contained in the DNS response it received. The resolver rejects the response if the two do not match and considers it a cache poisoning attempt. This assumes that the same key was used when encrypting the normalised version of the query and response messages.

Clearly, the additional entropy introduced by this encoding technique depends on the number of letters contained in the domain name being queried. For example, `0x20`-Bit encoding would add only three more bits of entropy to DNS queries for "123.net". The following shows the success probability of poisoning the cache of a resolver when implementing the protocol:

$$P = \frac{n}{N \cdot U \cdot T \cdot B}, \tag{2.5}$$

where B is the number of characters in the domain name being queried. The other symbols are the same ones used in equation (2.1). According to [29], the protocol adds an average of 12 additional bits of entropy. The authors of [29] also claim that over 99.7% of all DNS servers they have analysed could support `0x20`-Bit encoding.

Google Public DNS resolution service implements `0x20`-Bit encoding. In a report published by Google[16], two major issues have been identified when implementing the protocol:

- Some ANSs would respond with different character cases than the ones in the DNS query.
- Some ANSs are case sensitive and would respond with `NXDOMAIN` due to case mismatch. Receiving a `NXDOMAIN` RR indicates to the resolver that the name being queried does not exist in the DNS zone served by the queried ANS.

To overcome these issues, Google created a whitelist containing ANSs that would support `0x20`-Bit encoding. Only queries to ANSs in this whitelist are encoded. According to Google, the whitelisted ANSs account for more than 70% of Google's DNS traffic.

2.4.3 WSEC DNS

Wildcard secure (WSEC) DNS increases the entropy of DNS queries by randomising names contained in newly introduced `TXT` and `CNAME` RR queries. These queries are sent by resolvers to ANSs that support the protocol. Additional `TXT` and `CNAME` RRs are created in DNS zones served by ANSs to support WSEC DNS. The administrator of an ANS is expected to perform the following tasks to make use of WSEC DNS:

- Add two specific `TXT` RRs to the configuration of the DNS zone to indicate to resolvers that WSEC DNS is supported by this ANS for this zone. Figure 2.6

[16]`https://developers.google.com/speed/public-dns/docs/security.html`

```
;; Respond to DNS TXT queries and indicate that WSEC DNS is supported
;; for this zone.
*                   86400 IN TXT |wsecdns=enabled|
_test_._wsecdns_    86400 IN TXT |wsecdns=enabled|
```

Figure 2.6: `TXT` RRs added for WSEC DNS.

```
;; Respond with a CNAME RRs to requests coming from WSEC-capable resolvers
*._wsecdns_.www        IN CNAME www
*._test_.wsecdns_.www  IN CNAME _test_.wsecdns_
```

Figure 2.7: `CNAME` RRs added for WSEC DNS.

shows examples of RRs added to a zone's configuration file. The use of "`*`" indicates that this is a wildcard entry and hence the name of the protocol, *wildcard* secure DNS. An ANS responds to `TXT` queries, covered by this wildcard, with `|wsecdns=enabled|`.

- Add two `CNAME` RRs for *each* WSEC-protected host name. An example is shown in Figure 2.7.

WSEC DNS Process

Let us take the example when the host name being queried is "www.example.com" and when WSEC DNS is being used. The client sends a DNS query to its WSEC-capable resolver. Upon receiving this request, the resolver sends a specially formatted *discovery* DNS `TXT` RR query to the ANS serving the host name being queried. The query is in the form of ⟨`rand`⟩`._test_._wsecdns_.www.example.com`, where ⟨`rand`⟩ is a random alphanumeric string generated by the resolver. The ANS replies with a `TXT` RR containing `|wsecdns=enabled|` if it was configured for WSEC DNS.

The resolver now knows whether this ANS supports WSEC DNS for the host name being queried or not. If it does, then the resolver sends a specially crafted `A` RR query in the form of ⟨`rand`⟩`._wsecdns_.www.example.com`, where ⟨`rand`⟩ is another random alphanumeric string generated by the resolver and is the source of the added entropy. The ANS responds with a `CNAME` RR containing "www.example.com", an alias for the name being requested, ⟨`rand`⟩`._wsecdns_.www.example.com`. It also attaches the `A` RR for "www.example.com" to the same response.

There are a couple of reasons why WSEC DNS is unattractive. First, changes are required on every participating ANS and DNS resolver. The large number of DNS servers and the fact that ANSs are decoupled from resolvers make this a very difficult task to accomplish. Second, a WSEC DNS look-up would require two round trips instead of one, with larger query and response message sizes when compared to standard DNS query and response messages, increasing the amount of DNS traffic generated by ANSs and resolvers. We are not aware of practical DNS implementations that make use of WSEC DNS.

2.4.4 DNSSEC

The protocols we have described so far, other than 0x20-Bit encoding that can use AES to randomise the case pattern, do not deploy cryptographic controls to secure DNS. Examples of DNS security protocols that make significant use of cryptography include DNSSEC [1], Transaction SIGnature (TSIG) [103] and DNSCurve[17] . Out of the three protocols, DNSSEC is by far the most prominent in terms of acceptance, industry support and deployment. The use of DNSSEC requires changes to resolvers and ANSs. DNSSEC provides data origin authentication and integrity services to DNS using digital signatures [10]. It also provides means of public key distribution needed for the operation of the protocol. DNSSEC relies on resolvers verifying the authenticity of responses received from ANSs to counter DNS poisoning attacks. DNSSEC is supported by most of today's commercial and open source DNS implementations. For example, in Google Public DNS, a free Internet-based DNS service, support for DNSSEC came in January 2013.

Enabling DNSSEC on resolvers requires assigning an initial DNSSEC trust anchor. This is generally performed outside DNSSEC. DNSSEC Look-aside Validation (DLV), a protocol implemented today by a number of DNS packages, can be used for this purpose.

Definition 2.6. A DNSSEC trust anchor is a public key that acts as the entry point for a chain of authority.

The signed DNS root would make an ideal trust anchor to consider since it gives the receiver access to all possible chains of trust used when validating a DNSSEC response. DNSSEC deploys cryptographic measures to ensure the authenticity of the DNS data exchanged by digitally signing the DNS records. RSA and SHA-1 are to be used to sign the DNS records [12, 89]. When an ANS hosting a signed zone receives a DNS query, it responses with two RRs: the RR being queried and an RR for the digital signature associated with queried RR. DNSSEC introduces a number of new RRs to contain digital signatures, public keys and key hashes [10]:

[17]`http://www.dnscurve.org`

- `DNSKEY`: A record containing the public key for the zone. `DNSKEY` can contain either a Zone Signing Key (ZSK) or a Key Signing Key (KSK). The ZSK is used to sign the RRs in a zone, while the KSK is only used to sign the `DNSKEY` RR.
- Resource record signature (`RRSIG`): A record holding the signatures for a specific record type.
- Delegation signer (`DS`): A record that is submitted to the zone's parent. `DS` RRs are included only in the parent's zone, and correspond to `NS` RRs providing a link between the parent and the zone. `DS` RRs are part of the chain of trust from the zone's parent to the zone.
- Next secure (`NSEC`): A record used to provide proof of non-existence of a RR. Each name in a zone has an `NSEC` RR added when signed to allow both positive answers and negative answers to queries to be cryptographically secure.
- Next secure version 3 (`NSEC3`): This record is used in negative answers to prove that a name does not exist. It is similar in function to the `NSEC` record, but has some advantages in certain situations. Zones signed with `NSEC` are "walkable." This means that the entire contents of a zone can be retrieved simply by following the `NSEC` chain. Also, every name within a zone must be signed and have `NSEC` records. `NSEC3` uses cryptographic hashes to prevent zone walking while retaining the ability to prove negative answers.

The DNSSEC specifications were first published in 1999 in RFC 2535 [1]. This RFC was then obsoleted by RFC 4033 [10] in 2005. A number of supporting RFCs such as [12, 11, 34, 59, 45, 89] have also been published for DNSSEC.

Eleven years after the release of RFC 2535, in June 2010, the Internet Corporation for Assigned Names and Numbers (ICANN)[18] held the first Key Signing Key (KSK) ceremony event for the root's zone. DNSSEC for the root zone is a joint effort between ICANN and VeriSign, with support from the U.S. Department of Commerce. ICANN, a non-profit corporation, maintains an updated database[19] about the global deployment of DNSSEC. As of 15th of July 2013, there were:

- 317 TLDs in the root zone in total, 111 of which have been signed. Examples of signed TLDs include ".uk", ".com" and ".org".
- 107 TLDs having trust anchors published as `DS` records in the root zone.

[18] http://www.icann.org
[19] http://stats.research.icann.org/dns/tld_report

- Only 3 TLDs having trust anchors published in the Internet Systems Consortium (ISC) DNSSEC Look-aside Validation (DLV) repository.

Despite the fact that the DNS root, ".", was signed in 2010, DNSSEC has not gained the expected momentum or adoption levels, at least in the last three years. This is clearly reflected in the numbers published on ICANN's web site[20], where for example only three TLDs have trust anchors published in the ISC DLV repository. An Asia-Pacific Network Information Centre (APNIC) report[21] shows that in February 2013, around 3.3% of users had DNSSEC validating resolvers. In May 2013, this value rose to 8.1%, a 4.8% increase. This increase has been mainly attributed to the Google Public DNS service fully enabling DNSSEC. According to the APNIC report: "*Of the 2.34M unique IP addresses of clients who ran this experiment, we saw 174,082 clients use Google Public DNS servers, or 7.4% of all tested clients*". These numbers relate to DNSSEC on resolvers; they do not provide an overview of how many ANSs and zones have DNSSEC configured. Although DNSSEC has had a slow start, we believe that the adoption level will pick up over time. The adoption of DNSSEC by the Google Public DNS servers and other major Internet service providers will further push in this direction. The adoption of DNSSEC is being tracked by SecSpider[22], a DNSSEC monitoring project that is sponsored by VeriSign.

2.5 DepenDNS

2.5.1 Introduction to DepenDNS

DependDNS [97] is proposed as a client-based DNS implementation designed to protect clients from cache poisoning attacks. The fundamental concept behind DepnDNS is sending the same DNS query to multiple resolvers and then evaluating the responses. The evaluation is based on an algorithm referred to as π in [97]. DepenDNS is supposed to be practical, efficient and secure, according to [97].

The authors of [97] position DepenDNS as a comprehensive solution against cache poisoning attacks. Therefore, the protocol should be able to protect clients from various DNS cache poisoning attacks including the following three generic spoofing attack scenarios:

- **Scenario 1:** A spoofing attack against a client in which the attacker sends spoofed DNS replies to the client. We assume that the attacker has no access to

[20]http://stats.research.icann.org/dns/tld_report
[21]http://labs.apnic.net/blabs/?p=368
[22]http://secspider.cs.ucla.edu/growth.html

the DNS requests and hence is not aware of the host names being requested. Let us suppose that this attack has a success probability of p_1 when DepenDNS is not deployed.

- **Scenario 2:** A spoofing attack against the DNS resolver. We assume that the attacker has no access to DNS requests and hence is not aware of the host names being requested. In this scenario, the attacker tries to poison the resolver's DNS cache for an arbitrary host name at any point in time. Let us assume that the attack has a success probability of p_2. The impact of this attack is higher compared to scenario 1 since it would affect all clients served by the targeted resolver when requesting entries that have been poisoned.

- **Scenario 3:** The attacker has control over the DNS resolver and hence has visibility of the DNS requests. The probability of success of a spoofing attack is 1 when DepenDNS is not deployed.

The reader can think of the above scenarios from an abstract point of view, in which the exact implementation is irrelevant. For example, a random 16-bit TXID may or may not be in use. The objective of using this approach is to evaluate the effectiveness of the DepenDNS protocol regardless of the underlying implementation. In addition, the spoofing approaches discussed above can also be used to conduct other types of attacks than cache poisoning, as we demonstrate in this section.

When DepenDNS is deployed, clients should be able to detect and prevent the above three generic attacks and hence decrease their success probabilities to a minimal value. This is supposed to be achieved by querying multiple DNS resolvers and evaluating the responses against a set of pre-defined conditions.

The fundamental security objective of DepenDNS is to protect clients from bogus IP addresses sent by DNS resolvers. These bogus IP addresses would have either arrived from an already poisoned resolver's cache or as a result of a spoofed DNS response message directed against the client. The former is the more likely scenario. An attacker would target a DNS resolver serving a large number of clients in order to achieve a higher impact. The protocol proposed in [97] describes how a client running DepenDNS can detect and reject such bogus IP addresses.

The protocol relies on forwarding the same DNS query to multiple resolvers and then evaluating the replies using an algorithm π. Algorithm π runs on the client's machine and accepts or rejects each IP address suggested by the resolvers. The decision is based on comparing a number of parameters against a set of pre-defined thresholds. Accepted IP addresses are passed to the client and are saved in a history table maintained by

DepenDNS. The resolvers and authoritative servers are not involved in the IP addresses selection process carried out by DepenDNS.

The concept of invoking multiple resolvers in the DNS resolution process has been proposed by other DNS security protocols such as DoX [108], described earlier in Section 2.4.1. The motivation behind using multiple resolvers is that the probability of poisoning several resolvers at the same time should be much lower than that of poisoning one resolver. Although DepenDNS makes use of the same approach, the parameters defined and the calculations carried out by its algorithm π are different.

Our work focuses on evaluating the security and deployability aspects of DepenDNS. Our approach consists of analysing the proposed protocol, investigating the existence of vulnerabilities and eventually attacking the protocol. First, we start with an explanation of the operation of DepenDNS and how algorithm π's calculations are performed. The reader will find that our explanation of algorithm π and the symbols we use differ slightly from the original DepenDNS paper [97]. Our aim is to give the reader a concise and clear description of the algorithm. Second, we provide a review of the protocol and highlight a number of unclear assumptions made in [97]. We also consider a number of practical deployment issues that should have been addressed in [97]. Third, we analyse if DepenDNS is vulnerable to cache poisoning, Denial of Service (DoS) and amplification attacks. We have discovered scenarios in which we were able to successfully exploit the protocol. The attacks that we have performed against DepenDNS are based on a full implementation of the protocol and the use of real data collected over a period of time. This data was collected using our implementation of the protocol. DNS responses received from public Internet DNS servers, in response to DNS queries that we generate, are evaluated by our implementation of the protocol. Further information about out experimental results are provided in later in the chapter. We clearly highlight any assumptions made for our attacks to be successful. Fourth, we study the performance and accuracy of DepenDNS.

2.5.2 DepenDNS Algorithm π

The decision making process of DepenDNS is carried out by algorithm π. This algorithm expects the following inputs [97]:

- The IP addresses returned by all the resolvers being queried for the given host name.
- Access to a history table containing the previously accepted IP addresses for the given host name. This is a separate table that is maintained by DepenDNS and

is different from the client's DNS cache table. In Section 2.5.3 we further analyse the suggestions made in [97] on how to build and maintain the history data.

The output of algorithm π is a set of IP addresses that are supposed to be legitimate for the host name being requested. The output is used to update the history table and is forwarded to the requesting entity on the client's machine. An example of a requesting entity is a web browser. It is important to highlight that DependDNS does not maintain history information about *rejected* IP addresses. Algorithm π defines the following parameters:

- t is the number of the resolvers to which the client is configured to send its DNS request messages.
- R_j is the set of IP addresses returned by the j^{th} resolver, where $1 \leq j \leq t$. We write $R_j = \{IP_{1_j}, IP_{2_j}, ..., IP_{l_j}\}$, where l_j is the number of distinct IP addresses returned by the j^{th} resolver. In practice, a DNS reply may contain duplicate IP addresses. DependDNS normalises the reply by removing the repeated IP addresses and including a single copy of each IP address in R_j.
- R is the set that contains all the distinct IP addresses in the replies from t resolvers, i.e. $R = R_1 \cup R_2 \cup ... \cup R_t$. We write $R = \{IP_1, IP_2, ..., IP_m\}$ where m is the number of the distinct IP addresses returned by the t resolvers.
- n_j^i is a variable that is set to 1 if $IP_i \in R_j$ and 0 otherwise.
- n^i is the number of times IP_i appears across all R_j, i.e. $n^i = \sum_{j=1}^{t} n_i^j$.
- $n^{\max} = \max(n^1, n^2, ..., n^m)$.
- $c_{current}^k$ is a variable with value between 0 and 1. The value of $c_{current}^k$ is calculated by dividing the number of occurrences of IP addresses in all R_j that share the same leftmost 16 bits (represented by the integer k) by the total number of IP addresses returned by the t resolvers. IP addresses that share the same leftmost 16 bits are considered to be part of the same class, k.
- H is the history data maintained by DependDNS. H contains the IP addresses that have been accepted by algorithm π for each host name.
- $c_{history}^k$ is a variable parameter with value between 0 and 1. The value of $c_{history}^k$ is calculated in a similar way as $c_{current}^k$ but uses the information in H for the host name being requested as input for calculation.

- A is the set of IP addresses that are accepted by algorithm π for a specific DNS request. Each run of algorithm π generates a new A.

In general, algorithm π makes the decision to accept or reject an IP address after examining data related to the number of occurrences, history information and the leftmost 16 bits of that IP address. For each address IP_i, algorithm π calculates the variables α_i, β_i, and γ_i as follows:

1. α_i can be thought of as an indicator for the distance between n^i and n^{max}. α_i is determined by comparing n^i to n^{max} along with a tolerance variable that is set to 20% in [97]. Specifically, we have:
$$\alpha_i = \begin{cases} 1, & \text{if } n^i \geq (0.8 \cdot n^{max}); \\ 0, & \text{otherwise.} \end{cases}$$
2. β_i is related to the history data of DepenDNS. β_i is set to 1 if the IP address for the host name under evaluation exists in H. This indicates that the IP address has passed the evaluation process at some earlier point in time. Thus:
$$\beta_i = \begin{cases} 1, & \text{if } IP_i \text{ exists in the history data, } H; \\ 0, & \text{otherwise.} \end{cases}$$
3. γ_i is related to the leftmost 16 bits of the IP address and is determined by comparing $c^k_{current}$ and $c^k_{history}$. γ_i is set to 1 if the absolute difference between $c^k_{current}$ and $c^k_{history}$ is at most 0.1.
$$\gamma_i = \begin{cases} 1, & \text{if } IP_i \text{ belongs to the } k^{th} \text{ class and} \\ & -0.1 \leq c^k_{current} - c^k_{history} \leq 0.1; \\ 0, & \text{otherwise.} \end{cases}$$

Once α_i, β_i, and γ_i are calculated for each IP_i, algorithm π constructs the following sets:

- R_α, which contains the IP addresses in R with $\alpha_i = 1$. Thus $R_\alpha = \{IP_i \in R : \alpha_i = 1\}$
- R_β, which contains the IP addresses in R with $\beta_i = 1$. Thus $R_\beta = \{IP_i \in R : \beta_i = 1\}$
- R_γ, which contains the IP addresses in R with $\gamma_i = 1$. Thus $R_\gamma = \{IP_i \in R : \gamma_i = 1\}$

Algorithm π then calculates N, which is referred to as the dispersion strength in [97]. N is calculated as follows:

$$N = \frac{|R_\alpha \cup R_\beta \cup R_\gamma|}{\text{Mode}(|R_1|, |R_2|, ..., |R_t|)}. \tag{2.6}$$

Upon calculating N, algorithm π proceeds to calculate the grade, G_i, for each address IP_i. The value of G_i determines whether an IP address is accepted or not. G_i is calculated as follows:

$$G_i = \alpha_i \cdot (G_\alpha - 10 \cdot (N-1)) + \frac{1}{2} \cdot (\beta_i + \gamma_i)(G_{\beta\gamma} + 10 \cdot (N-1)), \tag{2.7}$$

where G_α and $G_{\beta\gamma}$ represent the weights given to α and $\beta\gamma$ and are set to 60 and 40 respectively in [97]. Note that N is the only variable in the above equation, since the values of G_α and $G_{\beta\gamma}$ are fixed. IP addresses with grades higher than or equal to 60 are accepted and are used to update A and H.

2.5.3 Protocol Review

According to [97], a good percentage of end-points should be able to make use of DepenDNS, since it is intended to be a client-based protocol. This may result in a large number of clients running DepenDNS. Such a potential large deployment of a protocol should not only consider the security aspects of the protocol but should also study the deployment challenges and the expected practical impacts.

In the previous section we described the calculations performed by algorithm π along with the decision making process. To reach a decision on whether to accept or reject an IP address, the protocol makes a number of explicit and implicit assumptions. Unfortunately, some of these assumptions in [97] have not been justified or backed by supporting information. In addition, the protocol has not addressed some important operational aspects of DNS. In this section we examine the validity of some of the assumptions made in [97]. We also highlight some of the characteristics of DNS that the proposal in [97] has failed to recognise.

System Initialisation:

A client running DepenDNS needs to be configured with the IP addresses of the DNS resolvers that it needs to query. The method by which the resolvers' IP addresses are set on the client is not discussed in [97]. This might seems to be a minor issue but we believe that it has a great operational impact in the case of large scale deployments. Manual configuration of the resolvers' IP addresses is impractical and hence an automated IP assignment approach should be considered.

Managing the History Data of DepenDNS:

Variables that are related to the history data maintained by DNS have a significant influence on the decisions made by the protocol. The values of β_i and γ_i are determined by the existence of history data and are part of the grade calculation. The values of β_i and γ_i are set to 0 in case no information exists in the history for the host name being requested. This clearly introduces an operational challenge since the history is expected to be empty initially. To address this challenge, [97] proposes either deploying a centralised database that can be used when new domains are queried, or adjusting the values of G_α and $G_{\beta\gamma}$ accordingly. The first option is unrealistic and cannot be practically deployed: there are more than 233 million domain name registrations[23] and there exists no centralised database that contains information about all the domains; even if one existed, it would be impossible to maintain such a database and track changes in domains' information around the world. The second option can be implemented. However, the proposed changes in the values of G_α and $G_{\beta\gamma}$ are not included in [97] and there is no assurance provided that such changes will not negatively impact the security properties of the protocol.

Tolerance Value Used to Calculate α:

The value of α_i is set to 1 if $n^i \geq (0.8 \cdot n^{max})$ for IP_i. Else, α_i is set to 0. The use of a 20% tolerance level is not justified, nor is any guideline on how to select a suitable value given in [97]. We would have expected more detailed discussion of how to select such critical system parameters.

Tolerance Value Used to Calculate γ:

The value of γ_i is set to 1 if $-0.1 \leq c^k_{current} - c^k_{history} \leq 0.1$ for IP_i. Else, γ_i is set to 0. As with α, the use of a 10% tolerance level for γ should have been justified.

Class Consideration by γ:

Algorithm π determines the value of γ_i based on the leftmost 16 bits of IP_i. The authors of [97] claim that a domain name may have several IP addresses but these IP addresses usually share the same leftmost 16 bits. However, no evidence or experimental data to support such a claim is offered in [97].

[23] http://www.verisigninc.com/assets/domain-name-brief-july2012.pdf

Number of Resolvers:

The proposed implementation of the protocol considers the use of 20 resolvers. However, the proposal does not explain the reasons behind choosing this number of resolvers. We use this number when analysing the protocol's behaviour in the coming sections. On the other hand, corporate networks generally deploy a small number of resolvers internally, typically 2 or 3. Adding 20 resolvers for the sake of implementing DependDNS is clearly an expensive exercise. Although service providers deploy a larger number of servers compared to corporate entities, only few might employ such a number of resolvers. This introduces another challenge, which is the method through which the DNS resolvers are selected.

2.5.4 Attacking DependDNS

The implementation of DependDNS is supposed to provide a good level of protection against DNS cache poisoning attacks. In Section 2.5.1 we referred to three attack scenarios that clients running DependDNS should be able to detect and prevent. Each attack has its own probability of success. In this section, we explore how the protocol behaves under a number of conditions with the intention of trying to find and exploit vulnerabilities in the protocol. We were able to find conditions under which we can poison the cache of DependDNS, perform a denial of service attack against the protocol, and execute amplification attacks that can trigger the generation of high volume of network traffic. Our cache poisoning and DoS attacks show that implementing DependDNS has no effect in lowering the probability of success of the three attack scenarios identified in Section 2.5.1. We state any assumptions we make for our attacks to be successful.

General Assumptions

Our general assumptions are as follows:

Assumption 2.1. *The attacker knows the IP address of one of the t resolvers that the client communicates with.*

Assumption 2.2. *The attack is bounded to a single resolver.*

This assumption is made to make our attack model realistic and also considers a worst case scenario for the attacker: If an attack against DependDNS is successful when a message from a single resolver is bogus, then it will certainly be successful when two or more resolvers are targeted.

Assumption 2.3. *The client is configured to use 20 resolvers as suggested in [97], i.e we set $t = 20$. Our attacks can still be successful for other values of t.*

The above general assumptions apply across all our attacks. However, some of the attacks we conduct might require meeting extra conditions. We will clearly highlight any additional assumptions we make.

DNS Cache Poisoning Attack

The fundamental security objective of DepenDNS is to protect clients from bogus IP addresses received from DNS resolvers. Detecting these bogus IP addresses is based on the calculations performed by algorithm π and the rejection is based on comparing the grade of each IP address to 60. IP addresses having grades G_i with $G_i \geq 60$ are accepted and are added to A and H. In this attack we attempt to circumvent the protocol by trying to achieve a grade of 60 or higher and eventually inject bogus IP addresses for a host name into the history data of DepenDNS.

Assumption 2.4. *The history table of DepenDNS contains IP addresses for the host name being requested.*

The attacker's goal is to bypass the security controls implemented by DepenDNS and have algorithm π accept false information in the form of bogus IP addresses. To achieve this, the attacker needs to spoof an IP address, IP_{bogus}, in a DNS response in such a way that the resulting grade, G_{bogus} calculated using equation (2.7), exceeds 60. Assumption 2.4 implies that the value of β is 0 for IP_{bogus}, and α_{bogus} is likely to be 0. The reason for this is that n^{bogus} is 1 since the attacker would target a single resolver as per Assumption 2.2, making it difficult for n^{bogus} to be above the threshold of n^{max}. This leaves the attacker with one variable, γ_{bogus}, to focus on. The attacker needs to make sure that γ_{bogus} for IP_{bogus} is 1. To achieve this, the following conditions must be met:

- The leftmost 16 bits of the bogus IP address are the same as the legitimate IP addresses for the host name, i.e. IP_{bogus} belongs to a valid k^{th} class IP address for the host name being requested.
- $c^k_{current} - c^k_{history}$ is within the pre-defined threshold, i.e. $-0.1 \leq c^k_{current} - c^k_{history} \leq 0.1$

Since the values α_{bogus} and β_{bogus} are 0, then the grade for IP_{bogus} can be calculated as

$$G_{bogus} = \frac{1}{2}(G_{\beta\gamma} + 10 \cdot (N-1)),$$

assuming γ_{bogus} is 1. For IP_{bogus} to be accepted, the value of G_{bogus} must be 60 or higher, i.e. the following condition must be met:

$$\frac{1}{2}(G_{\beta\gamma} + 10 \cdot (N - 1)) \geq 60.$$

Since $G_{\beta\gamma}$ is 40, the condition that $N \geq 9$ will guarantee that IP_{bogus} achieves the passing grade.

Our experiments have shown that the value of N is 1 or less for most host names. However, this is not the case when the host name is served by a CDN. We have noticed that the value of N is within a range that would allow attackers to inject bogus IP addresses using the technique we have explained in this section. For example, Figure 2.8 shows that the average value of N for "www.live.com" is 13.5. We have found similar results for other host names such as "maps.live.com", "www.youtube.com" and "www.vmware.com".

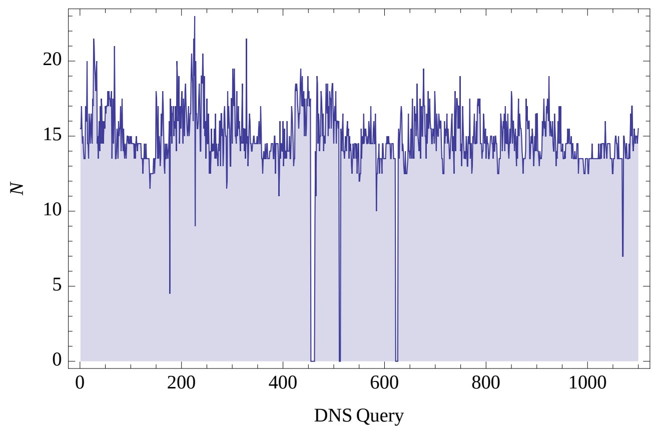


Figure 2.8: Value of N over time for "www.live.com". The reader would notice that the shows that the average value of N for "www.live.com" is 13.5. As described in Section 2.5.4, the condition that $N \geq 9$ will guarantee that IP_{bogus} achieves the passing grade, allowing DNS cache poisoning to succeed.

Our full experimental results are described in Section 2.5.5; meanwhile, Table 2.2 provides the percentage of runs when $N \geq 9$. A run is defined as the execution of algorithm π against R and H when a host name is being requested. The table shows that a high percentage of runs had $N \geq 9$ for the host names listed earlier. This gives the attacker an opportunity to launch her attack during the majority of runs. Please note that this is based on real data collected over time and hence includes situations when there are no response messages due to network connectivity issues, causing the value of N to be 0.

We have simulated the above attack by injecting a bogus IP address, 96.17.222.222,

Host name	Number of runs	% of runs with $N \geq 9$
www.live.com	1100	98.3
maps.live.com	1100	98.2
www.youtube.com	1100	98.5
www.vmware.com	1100	80.3
www.hsbc.com	1100	0

Table 2.2: Percentage of runs with $N \geq 9$.

into the cache of one of the resolvers for the host name "www.live.com". Our attack was successful, and the bogus IP address was accepted by our implementation of algorithm π and added to the history data.

Impact: *An attacker can inject a bogus IP address that points to a malicious website or inject IP addresses that can make the host being requested unreachable. The attack is applicable when the host name is hosted by a CDN and the client is running DepenDNS.*

Denial of Service Attack

Unlike a network-based Denial of Service (DoS) attack, our work targets the layer where DepenDNS would operate and where the decision of accepting or rejecting an IP address takes place. In our attack we try to force algorithm π into rejecting all IP addresses in R for a host name, hence making the host unreachable by clients running DepenDNS. The same spoofing attack scenarios listed in Section 2.5.1 can be used by the attacker with the same success probabilities of p_1, p_2 and 1 respectively.

Assumption 2.5. *The history data of DepenDNS does not contain information about the host name being requested.*

Consider a run of the algorithm π on a set of sets of returned IP addresses R_j, $1 \leq j \leq t$. The above assumption implies that the value of both β_i and γ_i is 0 for each IP_i in R. As a result, $R_\beta = \emptyset$, $R_\gamma = \emptyset$ and $G_i = \alpha_i \cdot (G_\alpha - 10 \cdot (N-1))$. For our attack to succeed, all IP_i in R should have a grade value, G_i, of less than 60. Therefore, the following condition must be met for each i:

$$\alpha_i \cdot (G_\alpha - 10 \cdot (N-1)) < 60.$$

Since G_α is known to be 60, then N must be higher than 1 for all IP_i to be rejected. In our situation, N can be calculated as:

$$N = \frac{|R_\alpha|}{\mathrm{Mode}(|R_1|, |R_2|, ..., |R_t|)},$$

since $R_\beta = \emptyset$ and $R_\gamma = \emptyset$.

To increase the value of N, an attacker would need to focus on increasing the size of R_α or decreasing the modal value of $|R_j|$. Decreasing the modal value proved to be very difficult since we assume that the attacker targets one resolver only (as per Assumption 2.2).

Our experimental results presented in Section 2.5.5 show the value of N for a number of host names queried over a period of time. For example, the average value of N for “www.live.com” is 3.5, meaning that the conditions for the DoS attack to succeed are met. Correspondingly, Figure 2.9 shows that no IP addresses for “www.live.com” were accepted during the vast majority of runs.

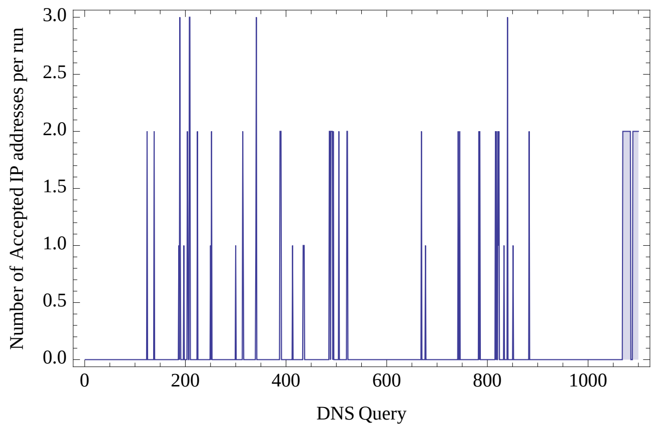
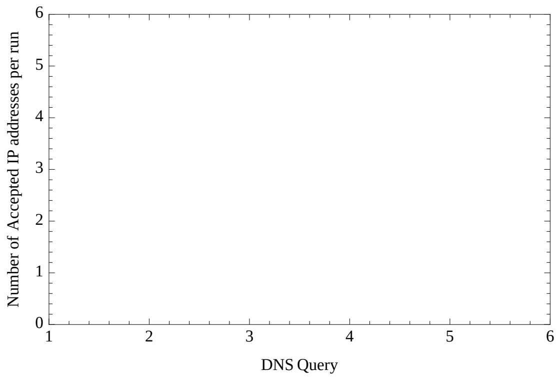


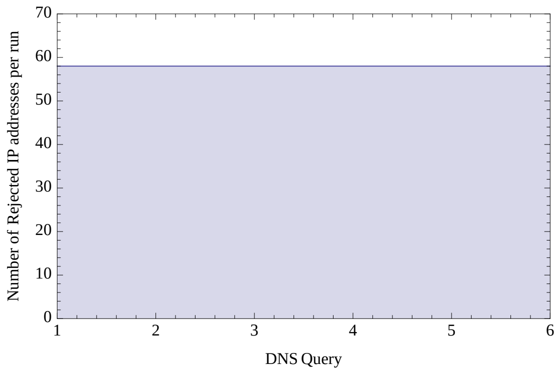
Figure 2.9: Number of IP addresses accepted over time for “www.live.com”. With an average value of N for “www.live.com” is 3.5, means that the conditions for the DoS attack to succeed are met. The figure shows that no IP addresses for “www.live.com” were accepted during the vast majority of runs.

On the other hand, we have noticed that the value of N can be easily influenced in the case when the host name being requested is hosted by a CDN. Therefore, rather than injecting bogus IP addresses in the DNS cache of a resolver, an attacker would include a good number of correct IP addresses for the host name. The goal is to maximise the number of IP addresses that pass the α test and hence increase the value of $|R_\alpha|$.

We simulated the attack using real data collected from querying 20 DNS resolvers (the proposed number of resolvers to query as per [97]) for “www.youtube.com” and we were able to force algorithm π into rejecting all IP addresses received from all 20 resolvers. We tested this for six consecutive runs and the attack was successful during each run. Before the attack, a total of six IP addresses would have been accepted (see



(a) Number of IP addresses accepted during six runs



(b) Number of IP addresses rejected during six runs

Figure 2.10: Results of the attack against "www.youtube.com". After injecting a number of valid IP addresses as per the technique described in this section we found that algorithm π starts rejecting all the IP addresses received from the 20 resolvers. The two figures show the number of accepted and rejected IP addresses during the six runs.

Section 2.5.5 for details). After injecting a number of valid IP addresses as per the technique described in this section we found that algorithm π starts rejecting all the IP addresses received from the 20 resolvers. Figures 2.10a and 2.10b show the number of accepted and rejected IP addresses during the six runs.

Impact: *An attacker can perform a DoS attack against a specific host name when it is hosted by a CDN and when the client is running DependDNS.*

Amplification Attack

In this attack we try to exploit the fact that DependDNS employs a number of resolvers, t. The success of an amplification attack relies on the ability of the attacker to trigger the generation of a large volume of traffic by sending requests of negligible size. The higher the amplification factor, the more severe the attack is. Such attacks are not new for DNS. In fact, DNS has been the target of various DNS amplification attacks [50], which rely on the fact that DNS response messages are significantly larger than response messages. In practice, an attacker will employ a set of machines under her control, like a botnet, to perform such attacks [50].

Assumption 2.6. *In our attack, we take the average size of a DNS request message to be 60 bytes and of a DNS response message to be 124 bytes. These numbers are based on the data collected during our experiments. This takes into account the TCP/IP headers.*

Our attack uses clients running DepenDNS and does not require the use of a botnet. We show how an implementation of DepenDNS can cause such attacks with a high amplification factor. We also show a sample code for performing the amplification attack.

Our attack is encoded in Hyper Text Markup Language (HTML) code. The sample code that we show in this section does not require the installation of any software on the client's machine and can be automatically executed by any application which can interpret HTML or JavaScript. The HTML code can be delivered to clients by email or can be published on a website which the client visits. To ensure a large scale effect, the attacker would publish this code on a popular website with thousands of concurrent visitors. Uploading the code onto social networking websites would be an attractive choice to the attacker. Figure 2.11 shows an example of HTML code that employs JavaScript. In the code in Figure 2.11, we use the image object, `img`, to force the web browser to perform a DNS look-up. The size of the above code is 306 bytes. The code generates two random strings, s1 and s2. These strings are then concatenated to build the host name in the image HTML tag, `img`. The attacker can change the number of host names being requested by increasing the length of the loops. In the above example, the variable "i" is incremented by one in every loop until reaching 10. Although changing the length of loops in the JavaScript to a higher value has a negligible effect on the size of the code, it has a significant impact on the amplification factor. For example, changing the length of the loops to 100 will increase the code size by only 1 byte, but will cause the generation of at least 736 kbytes of DNS request and response messages under Assumption 2.6. This number will be multiplied by the number of search domains the client is configured for. For example, the expected traffic will be at least 1.58 Mbytes if the client is configured for one search domain such as "example.com".

Impact: *Although, this attack applies to the standard DNS implementation, DepenDNS*

```
<html>
<head>
<meta http-equiv="refresh" content="5">
</head>
<body>
<script>
for (i=1;i<=10;i++)
 {
 s1= String.fromCharCode(97+Math.round(Math.random()*25));
 s2= String.fromCharCode(97+Math.round(Math.random()*25));
 document.write('<img src="ftp://'+s1+'.'+s2+'/f">');
 }
 </script>
 </body>
 </html>
```

Figure 2.11: Sample HTML code that employs JavaScript and could be downloaded and executed by the client's browser, resulting potentially in a DNS amplification attack, when DepenDNS is deployed.

amplifies it by a factor of 20 which makes it more attractive to attackers. Hence, an attacker can turn clients running DepenDNS into a source of a serious DoS attack. For example, an attacker could post this code to a popular website causing a storm of DNS traffic on the Internet.

2.5.5 Experimental Results

In this section we evaluate the operation of DepenDNS under a number of scenarios using real life data collected over a period of time. We queried 20 resolvers, all located in the US, for the following host names every five minutes, with a total of 1100 queries for each host name:

- "www.live.com". This host name has a `CNAME` RR of "a134.g.akamai.net" and is served by a CDN.
- "maps.live.com". This host name has a `CNAME` RR of "a1234.g.akamai.net" and is served by a CDN.
- "www.youtube.com". This host name has a `CNAME` RR of "youtube-ui.l.google.com" and is served by a CDN.
- "www.vmware.com". This host name has a `CNAME` RR of 'e508.g.akamaiedge.net'" and is served by a CDN.

- "www.hsbc.com".

We developed a perl script that implements DepenDNS. The script takes DNS response messages from the 20 resolvers and runs them through an implementation of Algorithm π. The script also maintains a history table as described in Section 2.5.2. We collected the following set of information for each host name listed above:

- The value of N for each run.
- The number of accepted and rejected unique IP addresses for each run.

A run is defined as the execution of algorithm π against R and H when a host name is being requested.

The results shown in this section validate the findings presented earlier in this chapter. We divide our experiments into two categories based on the availability of history information about the host name being requested.

Experimenting with no History Information

We present here the results of running DepenDNS without existing history information about the host name being requested. The results of all the runs show the following trends:

- A large percentage of valid replies are rejected by DepenDNS when the host name being requested is hosted by a CDN. For example, Table 2.3 shows that 98.6% of the unique IP addresses for "www.live.com" were rejected after 1100 runs.
- A large number of runs had no accepted IP addresses when the host name being requested is hosted by a CDN. During these runs, the host name being requested is considered unreachable by the client.
- IP addresses for host names that are not hosted by CDNs were accepted in all of the runs.
- The value of N varies depending on the host name being requested.

Table 2.3 shows the results of running DepenDNS against the five host names. We collected the following set of information for each host name that we evaluated:

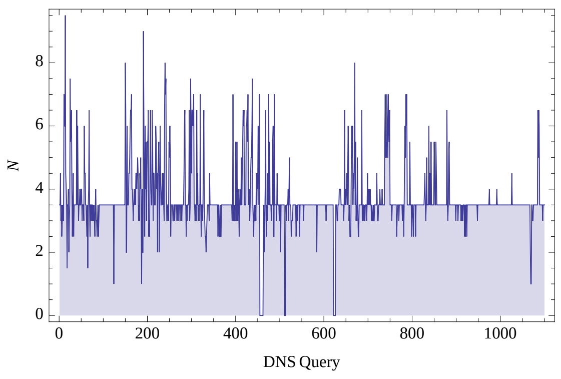
- The value of N for each run, calculated using equation 2.6.
- The number of accepted and rejected unique IP addresses for each run.

Host name	After run	Number of distinct accepted IP addresses	%	Number of distinct rejected IP addresses	%
www.live.com	1100	8	1.4	567	98.6
maps.live.com	1100	7	2.7	251	97.3
www.youtube.com	1100	6	5.2	110	94.8
www.vmware.com	1100	16	19.5	66	80.5
www.hsbc.com	1100	1	100	0	0

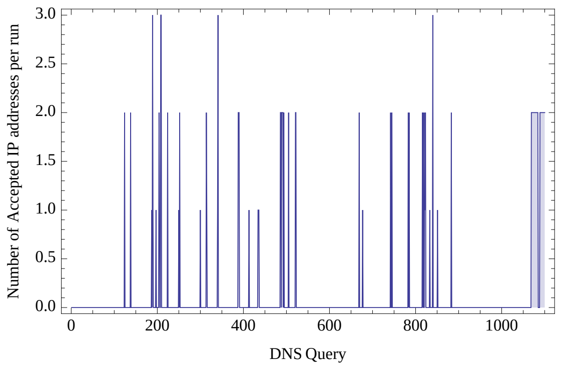
Table 2.3: Summary results for all host names without existing history information.

Figures 2.12, 2.13, 2.14, 2.15 and 2.16 show the value of N, the number of accepted IP addresses and the number of rejected IP addresses over time for the host names in Table 2.3. The figures show that when starting with an empty H, the DepenDNS history table, algorithm π rejects most of the IP addresses it receives for host names served by CDNs.

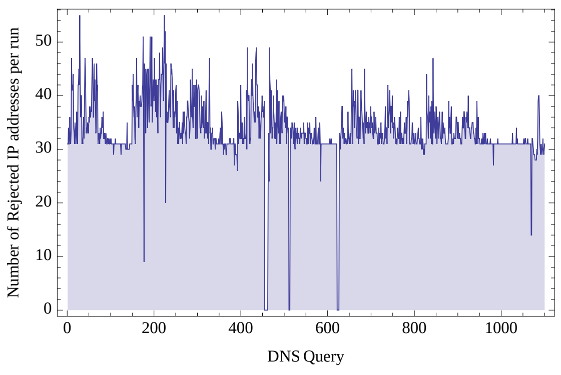
The reader might notice some dips in the graphs shown below. These are due to loss of network connectivity.



(a) Value of N over time

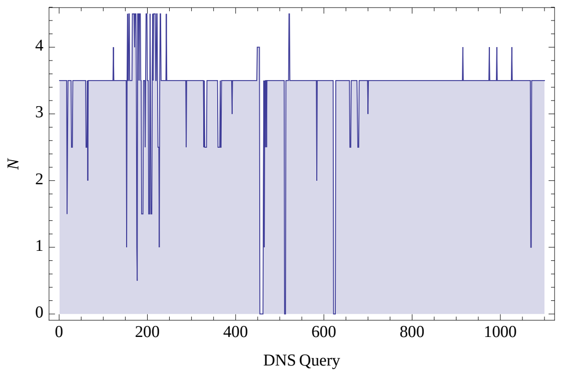


(b) Number of accepted IP addresses

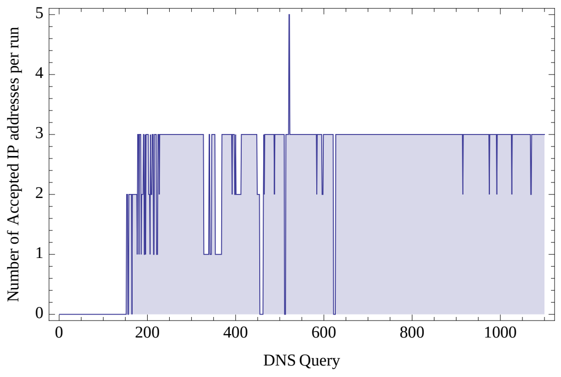


(c) Number of rejected IP addresses

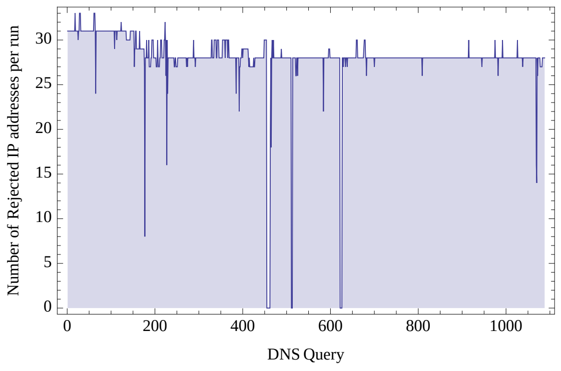
Figure 2.12: Results for “www.live.com” starting without existing history information.



(a) Value of N over time

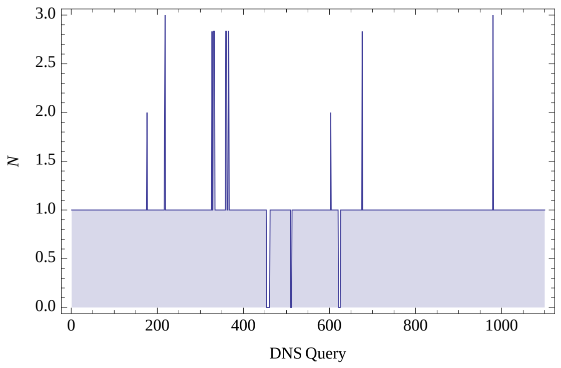


(b) Number of accepted IP addresses

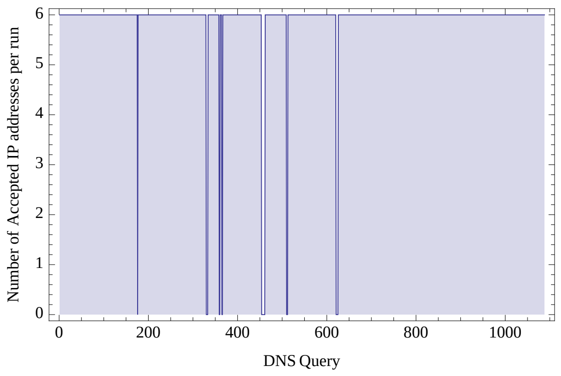


(c) Number of rejected IP addresses

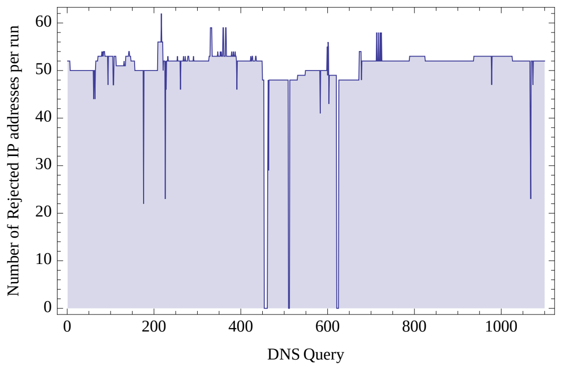
Figure 2.13: Results for “maps.live.com” starting without existing history information.



(a) Value of N over time

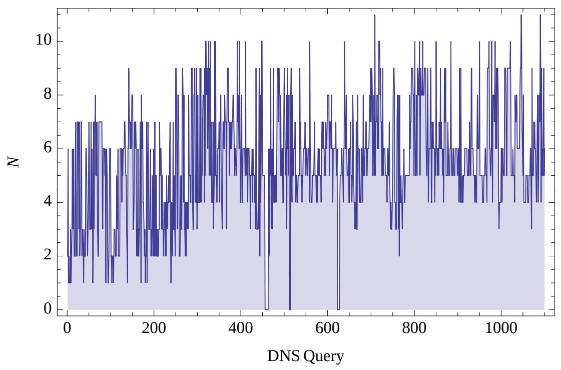


(b) Number of accepted IP addresses

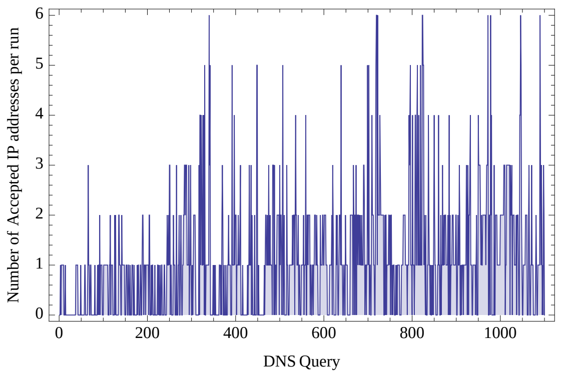


(c) Number of rejected IP addresses

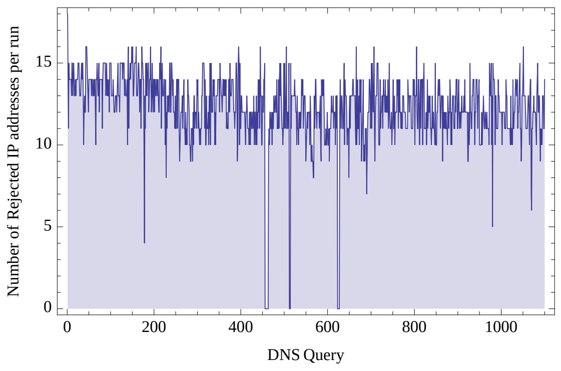
Figure 2.14: Results for “www.youtube.com” starting without existing history information.



(a) Value of N over time

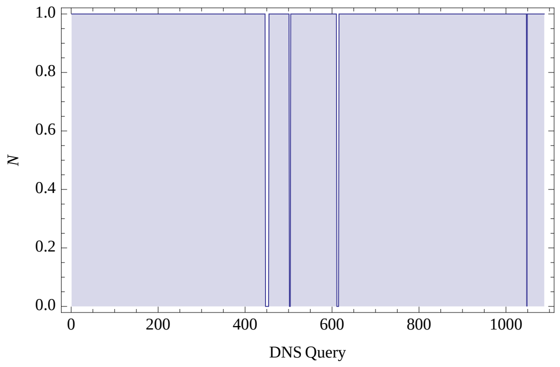


(b) Number of accepted IP addresses

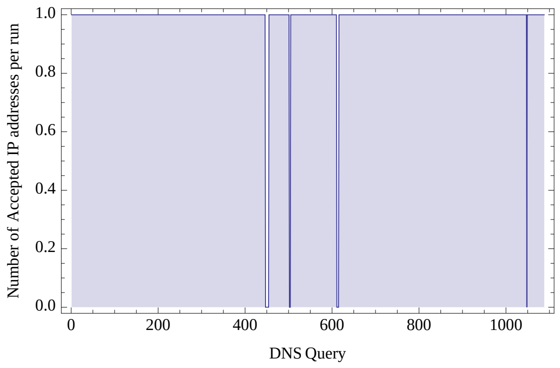


(c) Number of rejected IP addresses

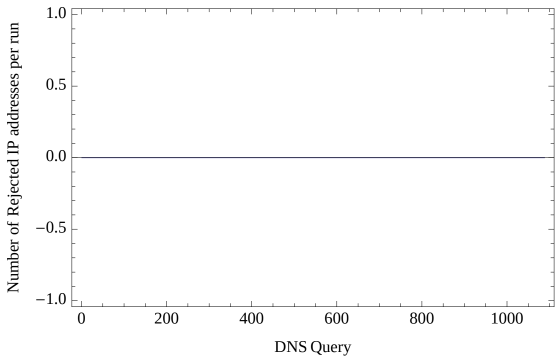
Figure 2.15: Results for “www.vmware.com” starting without existing history information.



(a) Value of N over time



(b) Number of accepted IP addresses



(c) Number of rejected IP addresses

Figure 2.16: Results for “www.hsbc.com” starting without existing history information.

Host name	Age of history	After run	Number of accepted IP address	%	Number of rejected IP address	%
www.live.com	1st run	1100	209	36.3	366	63.7
www.live.com	1 hour	1100	285	49.6	290	50.4
www.live.com	12 hours	1100	342	59.5	233	40.5
www.live.com	24 hours	1100	393	68.3	182	31.7
maps.live.com	1st run	1100	72	27.9	186	72.1
maps.live.com	1 hour	1100	105	40.7	153	59.3
maps.live.com	12 hours	1100	127	49.2	131	50.8
maps.live.com	24 hours	1100	156	60.5	102	39.5
www.youtube.com	1st run	1100	105	90.5	11	9.5
www.youtube.com	1 hour	1100	105	90.5	11	9.5
www.youtube.com	12 hours	1100	105	90.5	11	9.5
www.youtube.com	24 hours	1100	108	93.1	8	6.9
www.vmware.com	1st run	1100	47	57.3	35	42.7
www.vmware.com	1 hour	1100	63	76.8	19	23.2
www.vmware.com	12 hours	1100	70	85.4	12	14.6
www.vmware.com	24 hours	1100	74	90.2	8	9.8
www.hsbc.com	1st run	1100	1	100	0	0
www.hsbc.com	1 hour	1100	1	100	0	0
www.hsbc.com	12 hours	1100	1	100	0	0
www.hsbc.com	24 hours	1100	1	100	0	0

Table 2.4: Summary results for all host names with existing history information.

Experimenting with Existing History Information

In this section we evaluate DepenDNS when history information exists for the host name being requested. The data used to initialise the history of DepenDNS was collected at different points of time. We evaluate DepenDNS using history data collected in the following different ways:

- The first set of replies received from the t resolvers.
- The collection of replies received from the t resolvers after one hour.
- The collection of replies received from the t resolvers after 12 hours.
- The collection replies received from the t resolvers after 24 hours.

The results of all the runs show the following trends:

- A good percentage of valid replies are rejected by DepenDNS. The percentages are listed in Table 2.4 for the five host names we have queried.

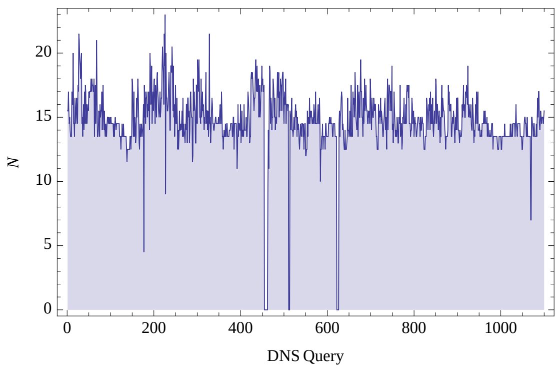
- The value of N is high for host names hosted by CDNs. For example the value of N is around 13.5 for "www.live.com" and 10 for "www.vmware.com".

We present the value of N over time along with the number of accepted and rejected IP addresses in each run for the host names that we have queried. The values presented are the results of running DepenDNS using as history the data collected from the first set of replies from the 20 resolvers. We collected the following set of information for each host name that we evaluated:

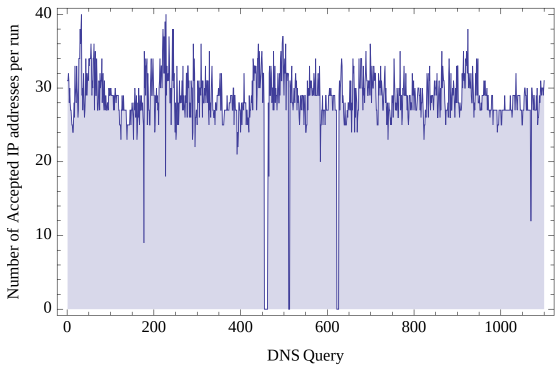
- The value of N for each run.
- The number of accepted and rejected unique IP addresses for each run.

Figures 2.17, 2.18, 2.19, 2.20 and 2.21 show the value of N, the number of accepted IP addresses and the number of rejected IP addresses over time for the host names in Table 2.4. When compared to the results reported in the previous section, the figures in this section show that when initialising H, in this case by populating it with the first set of replies from the 20 resolvers, algorithm π performs better and rejects less number of IP addresses received over time for host names served by CDNs.

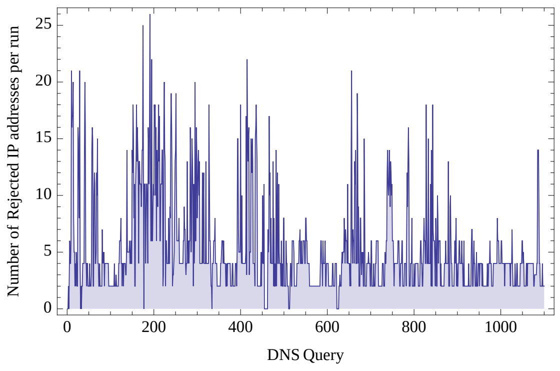
The reader might notice some dips in the graphs shown below. These are due to loss of network connectivity.



(a) Value of N over time

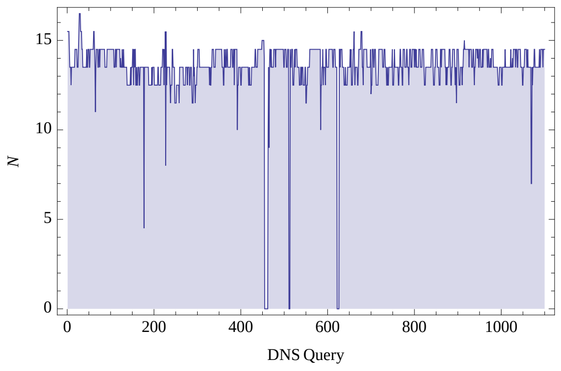


(b) Number of accepted IP addresses

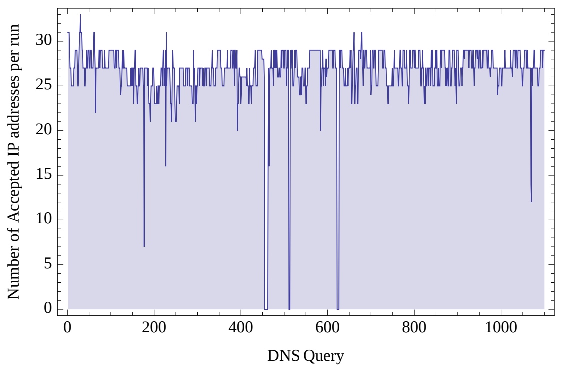


(c) Number of rejected IP addresses

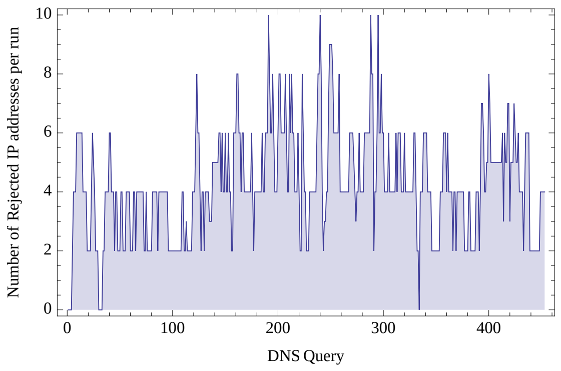
Figure 2.17: Results for “www.live.com” with existing history information.



(a) Value of N over time

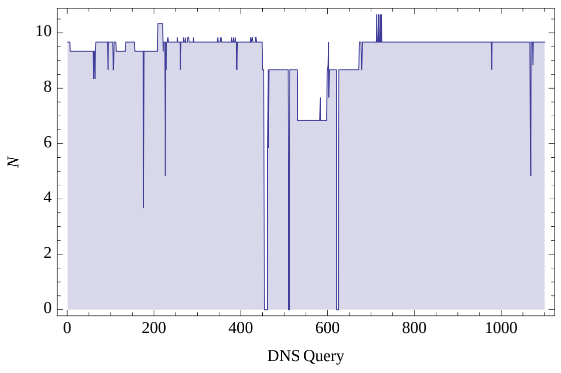


(b) Number of accepted IP addresses

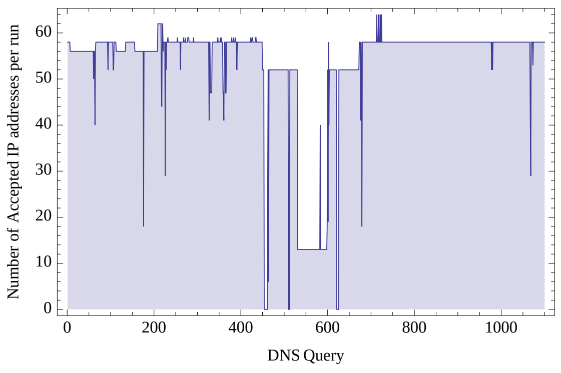


(c) Number of rejected IP addresses

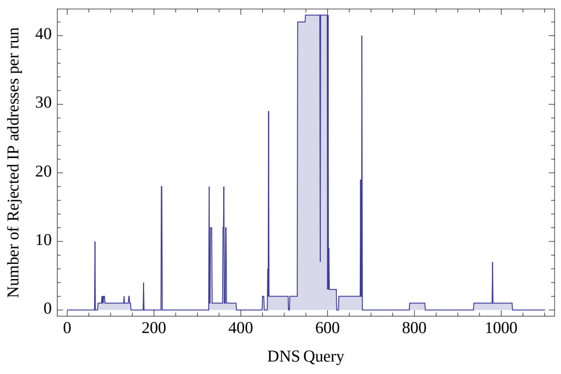
Figure 2.18: Results for “maps.live.com” with existing history information.



(a) Value of N over time

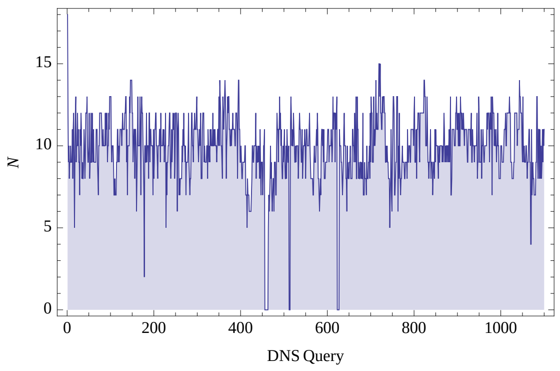


(b) Number of accepted IP addresses

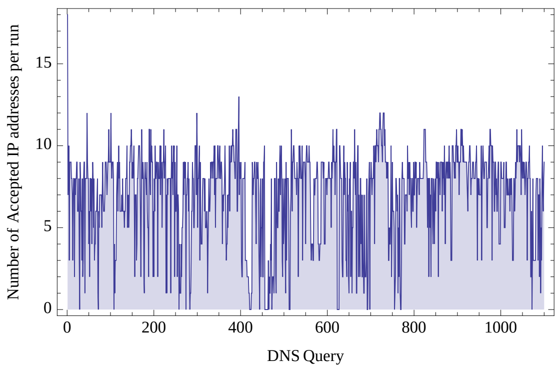


(c) Number of rejected IP addresses

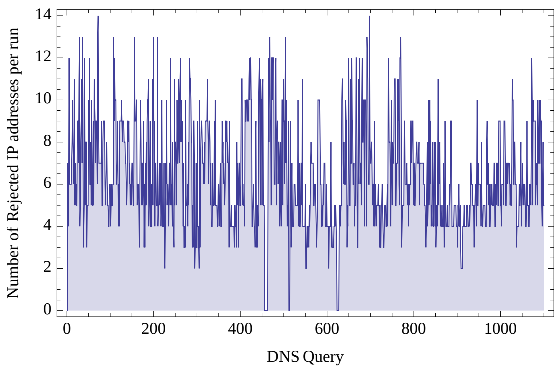
Figure 2.19: Results for "www.youtube.com" with existing history information.



(a) Value of N over time

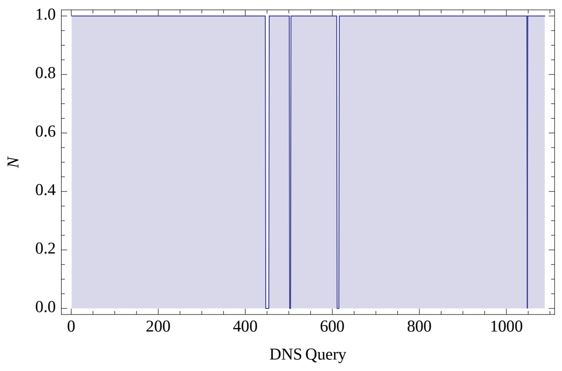


(b) Number of accepted IP addresses

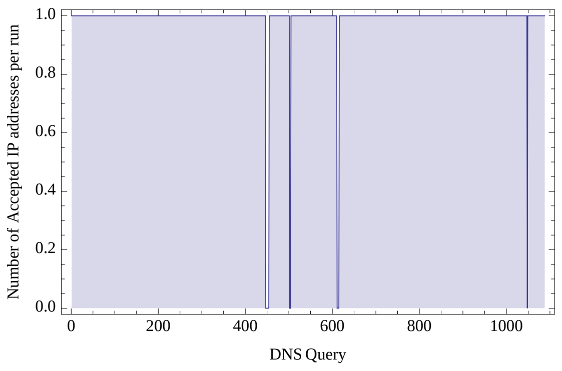


(c) Number of rejected IP addresses

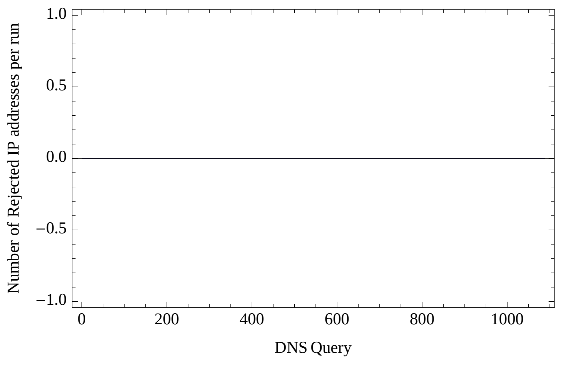
Figure 2.20: Results for “www.vmware.com” with existing history information.



(a) Value of N over time



(b) Number of accepted IP addresses



(c) Number of rejected IP addresses

Figure 2.21: Results for "www.hsbc.com" with existing history information.

Location of the t Resolvers

We have also conducted experiments that evaluate DependDNS when the t resolvers are distributed over multiple geographical locations. The objective was to compare the results of these experiments to the ones we have presented earlier. The overall results show a lower number of accepted IP addresses in each run when host names that are served by CDNs. Rejecting more IP addresses is largely attributable to how algorithm π calculates the value of N when the IP addresses it receives are served by resolvers consulting CDNs located in different countries or geographies, causing, in most cases, each resolver to reply with a different IP address based on the location (and configuration) of the CDN's DNS server. Host names that are not served by CDNs such as "www.hsbc.com" exhibited the same behaviour that we saw when using DNS resolvers located in the same geography.

2.6 Chapter Conclusion

DNS is critical to the operation of the Internet and hence maintaining the security of DNS is paramount. In this chapter we discussed a number of vulnerabilities in the protocol and its implementation, and how exploiting these vulnerabilities could impact the DNS service availability or integrity.

In summary, only a few of the proposed DNS security protocols have been adopted in practice; most of them have not been considered practical for deployment. The main reason behind this is the significant effort required to change the underlying DNS infrastructure to accommodate these new protocols. A good example is DNSSEC which is one of the most visible initiatives to secure DNS. Here, the challenges associated with the practical implementation of DNSSEC have lead to a significant delay in deploying the technology.

We also argue that proposals which attempt to address challenges in critical infrastructures should carefully study the impact of their implementations. For example, our analysis of DepenDNS has revealed a set of deficiencies in both the security controls and the operational aspects of the protocol. Although the protection controls implemented by DepenDNS do work for general web sites, domains that are hosted by CDNs have proven to be more of a challenge. The designers of DepenDNS made various assumptions, which were not justified or backed up by scientific evidence, for example the recommended number of resolvers to use and the DepenDNS history table population techniques. On the other hand, we have found conditions under which denial of service and cache poisoning attacks can be launched against DepenDNS. We have also shown that the implementation of DepenDNS can be exploited in an amplification attack. As a result, we do not recommend adopting DepenDNS with its current proposed design.

Chapter 3

TLS and DTLS

3.1 Introduction

In this chapter we provide the necessary background information and prerequisite material that are needed to establish an understanding of the TLS and DTLS protocols. First, we provide background information about the TCP/IP protocol suite and describe three fundamental networking protocols: IP, TCP and UDP. Second, we introduce Transport Layer Security (TLS), describe how the TLS protocol is structured and discuss its modes of operation. We also introduce Datagram Transport Layer Security (DTLS) and describe the differences between TLS and DTLS. We then present in detail the concept of padding oracles and show how an attack can be theoretically mounted against TLS using a padding oracle. Finally, we present a number of attacks against the TLS and DTLS protocols, serving as a forerunner to our attacks on DTLS and TLS, which we present later in Chapters 4 and 5.

3.2 The TCP/IP Protocol Suite

The Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite [22, 95], also known as the Internet protocol suite, is a set of networking protocols that are used on the Internet. The specifications of the protocol suite are managed by the Internet Engineering Task Force (IETF) and were first published by the Defense Advanced Research Projects Agency (DARPA) in RFC 791 [78]. This RFC was later updated by RFCs 1349 [9], 2474 [69] and 6864 [98]. The structure of the TCP/IP protocol suite consists of the following stack of layers, arranged from bottom-to-top: link, internet, transport and application. A networking protocol is mapped to one, or more, of these four layers. For example, the Hypertext Transfer Protocol (HTTP)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.1: Structure of the IPv4 header as per RFC 791. The structure shows the different fields that are available in an IPv4 packet. We descricbe some of these fields in 3.2.1

maps to the application layer.

Every layer in the TCP/IP protocol stack maintains its own datagram (envelope) structure and performs a number of operations such as fragmenting messages received from its upper-layer (if necessary), assembling datagrams received from its lower-layer (if necessary), and adding, verifying and removing its own datagram header. For example, a message received from an application, such as a browser is first fragmented (if necessary) by the transport layer before a fragment is encapsulated in a transport layer datagram that contains a transport layer-specific header. This datagram is then encapsulated in lower-layer (internet and link) datagrams as it propagates down the TCP/IP protocol stack.

3.2.1 The Internet Protocol

The Internet Protocol (IP) is the core protocol that facilitates the operation of the Internet. In this section, we provide a basic introduction to IP focusing on aspects related to the attacks discussed in Chapters 4 and 5.

IP operates at the internet layer of the TCP/IP protocol stack [22, 95] and has its own datagram (packet) structure. The structure of IPv4 [78] packet headers is shown in Figure 3.1. We will be referring to IPv4 only from now on in this thesis.

IP is a connection-less protocol, i.e. no connection is established at the IP layer between the two communicating hosts. In addition, no information regarding a transaction state is maintained by hosts at this layer. Other than the 2-byte `Header Checksum` field, shown in Figure 3.1, there is no error detection or control facilities built into IP. According to [78], the value of the 2-byte `Header Checksum` field is assigned by the sender to the 16-bit one's complement of the one's complement sum of all 16-bit bytes in the IP header [78]. For the purpose of calculating the checksum, the `Header Checksum` field is initially set to zero. Clearly, the `Header Checksum` must be calculated every time an IP datagram is created or a change is made to any of the other IP header fields. The receiver of an IP datagram is expected to also compute the value of `Header Checksum` for validation purposes. A receiver discards an IP datagram if the checksum it calculates is different from what the `Header Checksum` field contains. Obviously, using the `Header Checksum` field only provides weak integrity protection since the value is calculated using a basic keyless algorithm over known information. In practice, the `Header Checksum` field is mainly meant to detect changes resulting from network errors.

IP Spoofing

IP spoofing is an integral part of many attacks. It refers to creating a *valid* IP packet using a forged source IP address, typically with the objective of impersonating another IP host. Generally, the receiver cannot reliably identify if a packet has been spoofed using only information contained in the IP header. In this case, the receiver must rely on upper layer protocols (for example, TLS and DTLS) to detect IP spoofing.

3.2.2 The Transmission Control Protocol

The `Protocol` field in an IPv4 header contains the identification number of the transport protocol. For example, TCP is assigned to protocol number 6, while the User Datagram Protocol (UDP) is assigned to protocol number 17. The protocol identification numbers are managed by IANA [85].

The Transmission Control Protocol (TCP) [79] operates at the transport layer, just above the IP layer as shown in Figure 3.2. The specifications of TCP are given in RFC 793 [79]. Several RFCs [22, 80, 43, 42] were later published to update the TCP specifications given in RFC 793. TCP is a reliable connection-oriented protocol; a three-way TCP handshake must complete successfully before a connection is established between two hosts. TCP provides confirmation of reception through its acknowledgement facility. TCP segments are retransmitted by the sender if not acknowledged by the receiver within a timeout interval that is maintained by the sender. TCP sequence numbers along with a window are used to maintain the state of the connection. The window indicates the range of acceptable sequence numbers beyond the last successfully received one. The window is maintained by the receiver and its value is communicated to the sender inside every acknowledgment sent. Sequence numbers are also used for re-ordering incoming TCP datagrams and discarding duplicate TCP datagrams. The starting sequence number of a connection must be randomly assigned. Implementations of TCP must maintain separate state information for every TCP connection. A TCP connection progresses through a series of states during its lifetime [79]: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT and CLOSED .

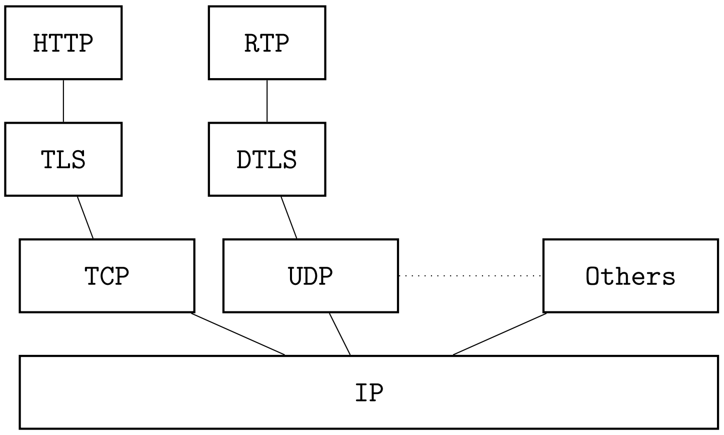


Figure 3.2: Relationships between protocols. The figure shows the construct in which protocols operate.

TCP maintains its own *basic* checksum field. According to RFC 793 [79] "*The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.*"

Spoofing a TCP datagram requires performing IP spoofing, described earlier, as well as guessing the correct 4-byte TCP sequence number and the 2-byte TCP source port, and updating the TCP `Checksum` field accordingly. The correct 4-byte TCP sequence number and the 2-byte TCP source port are readily available to a Man-in-the-Middle (MITM) or an attacker who has access to the TCP header information. The attacker also needs to make sure that his spoofed TCP datagrams arrive ahead of the legitimate ones. Otherwise, the spoofed datagrams would be discarded by the receiver on arrival. Recall that in a MITM configuration, the attacker can control the flow of information, and hence can delay, alter or drop legitimate datagrams.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgement Number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.3: Structure of the TCP header as per RFC 793. A number of TCP header fields shown in this figure, such as `Sequence Number`, `Acknowledgement Number`, `Window` and `Checksum`, and described in Section 3.2.2, are required to achieve the TLS reliability feature that upper layer protocols such as TLS rely on. We describe the relationship between TCP and TLS later in this chapter.

3.2.3 The User Datagram Protocol

Similar to TCP, the User Datagram Protocol (UDP) [77] is a protocol that operates at the transport layer of the TCP/IP protocol stack, as shown in Figure 3.2. The specifications of UDP are given in RFC 768 [77]. UDP is an unreliable connectionless protocol that provides best-effort delivery of datagrams, allowing applications to generate and send data at any time. The structure of a UDP datagram header is shown in Figure 3.4, a much simpler structure than the TCP header. The reason behind this simplicity is the minimal number of services that the UDP protocol provides when compared to TCP. For example, unlike TCP, UDP does not require the establishment of a connection before data is exchanged between to IP nodes. This UDP property would be suitable, and in many cases desired, for applications where reliability is not of a concern, DNS as one example.

Similar to TCP, UDP maintains its own a checksum field, `Checksum`. A UDP datagram is discarded by the receiver if the `Checksum` field is found to be invalid. The UDP `Checksum` field can be set to zero, indicating that the sender has not calculated

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.4: Structure of the UDP header as per RFC 768.

the checksum for the UDP datagram, and that the field can be ignored by the receiver; in this case, the receiver can choose to accept or reject the UDP datagram.

3.2.4 Implementing the TCP/IP Protocol Suite

Typically, operating systems would implement most of the TCP/IP protocol stack. This includes performing tasks related to handling datagrams for the internet and transport layers. Examples of these tasks include constructing datagrams, sending and receiving datagrams, and maintaining datagram queues. The exact implementation of the TCP/IP protocol suite is operating system-dependent. For example, in the case of the Linux kernel, there are two datagram buffers for each direction of traffic (incoming and outgoing traffic). The first buffer is located at the internet layer. The second buffer is located at the transport layer and its maximum size is based on the number of bytes the buffer can handle [26]. For example, the default maximum UDP buffer size for a Linux kernel is 131071 bytes[1]. Operating systems are expected to implement the standards' mandatory requirements when handling datagrams. For example, operating systems should handle incoming UDP datagrams in accordance to [95], "*If the incoming UDP datagrams arrive faster than the application can read them and if the queue fills to a maximum value, UDP datagrams are discarded by UDP. UDP will continue to discard UDP datagrams until there is space in the queue.*"

3.3 Cryptographic Primitives

TLS and DTLS make use of a number of cryptographic primitives. In this section, we give an introduction to cryptographic hash functions and message authentication codes (MACs). We will refer to these cryptographic primitives in the course of this chapter

[1]`http://access.redhat.com/site/documentation/en-US/JBoss_Enterprise_Web_Platform/5/html/Administration_And_Configuration_Guide/jgroups-perf-udpbuffer.html`

and the next two chapters, and describe how they are deployed in the context of TLS and DTLS.

3.3.1 Hash Functions

A hash function takes an input, M, a bitstring of arbitrary length and produces a message digest, x, that is of fixed length. A cryptographic hash function, h, is designed to withstand pre-image, second pre-image and collision attacks.

Definition 3.1. Pre-image resistance: For essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e. to find any pre-image x' such that $h(x') = y$ when given any y for which a corresponding input is not known.

Definition 3.2. Second pre-image resistance: It is computationally infeasible to find any second input which has the same output as any specified input, i.e. given x, to find a second pre-image $x' \neq x$ such that $h(x) = h(x')$.

Definition 3.3. Collision resistance: It is computationally infeasible to find any two distinct inputs x and x' which hash to the same output, i.e. such that $h(x) = h(x')$.

Hash functions are used in a wide variety of applications including the construction of other cryptographic primitives such as message authentication codes (MACs), digital signatures, pseudo random functions (PRFs) and pseudorandom number generators (PRNGs). Clearly, the security of these applications relies on the cryptographic strength of the underlying hash function used and its resistance to attacks. Widely used hash functions include MD5 [87, 99], SHA-1 [71] and SHA-256 [71]. SHA-3 [71], based on Keccak [20], is the latest addition to the list of hash functions. Today, it is practically feasible to find collisions in MD5 [107], and hence MD5 should *not* be considered for constructing other cryptographic primitives [99]. SHA-1 collision resistance has also been under attack [105, 106].

3.3.2 Message Authentication Codes

A MAC can be thought of as a fixed-size cryptographic tag, T, that is produced from applying algorithm f on a variable-length message, M, along with a secret key, K. The use of the secret key, K, in MAC allows the receiver to verify the integrity and authenticity of the received message. Integrity and authenticity verification is a feature that the use of hash functions, for example, cannot provide. Algorithm f can be constructed using block ciphers (for example, CBC-MAC [70], OMAC [47] and PMAC [109]), but can also be constructed using a cryptographic hash functions such as in the case of HMAC. The HMAC-based constructions [58] are the most commonly used MAC realisations and are the ones supported by the different TLS versions [31, 32, 33]. The different realisations of HMAC are denoted by HMAC-MD5, HMAC-SHA-1, HMAC-SHA-256, etc. We give more detail on the construction of HMACs in Chapter 5.

3.4 Introduction to Transport Layer Security

Transport Layer Security (TLS) is arguably the most widely used secure communications protocol on the Internet today. TLS and its predecessor, Secure Sockets Layer (SSL), establishes a secure channel between two parties. SSL was originally designed by Netscape and was never formally published as a *standard* by the IETF. Starting with SSL 2.0, a deprecated version that was released in 1994, Netscape's intent was to build a layer that can provide transparent protection services to a wide variety of application-level traffic. SSL 2.0 suffered a number of severe security flaws [104] and was shortly replaced by SSL 3.0, in 1995. Microsoft also responded to the flaws discovered in SSL 2.0 by creating their own protocol, called Private Communications Technology (PCT) [19, 21], also in 1995. PCT was not implemented by platforms other than Microsoft Internet Explorer and was eventually dropped by Microsoft. It is worth noting that a recent IETF standard was published to prohibit the use of SSL 2.0 [100].

All versions of SSL were mainly developed and maintained by Netscape; the IETF published RFC 6101 [38] for SSL 3.0, but for the historical record only. SSL was later adopted by the IETF and was specified as an RFC standard in 1999 under the name of TLS 1.0 [31]. It has since evolved through TLS 1.1 [32] to the current version TLS 1.2 [33]. Various other RFCs define additional TLS cryptographic algorithms and extensions such as the ones specified in [92, 90, 84, 52, 2, 93].

TLS is now used for securing a wide variety of application-level traffic: It serves, for example, as the basis of the HTTPS protocol for encrypted web browsing, it is used in conjunction with Internet Message Access Protocol (IMAP) or Simple Mail Transfer Protocol (SMTP) to cryptographically protect email traffic, and it is a popular tool to secure communication with embedded systems, mobile devices, and in payment systems.

TLS and SSL require an underlying reliable transport protocol to operate. This is mainly TCP, this being the dominant reliable transport protocol used on the Internet. TLS relies on TCP's reliability to reorder incoming datagrams and retransmit lost datagrams. An example of another reliable protocol that TLS can operate over is the Stream Control Transmission Protocol (SCTP) [72, 49]. DTLS, on the other hand, operates over an unreliable protocol such as UDP. We give more detail on DTLS in

Section 3.5.

TLS is actually a protocol suite, rather than a single protocol. The main component of TLS is the Record Protocol, which uses symmetric key cryptography (block ciphers, stream ciphers and MAC algorithms) in combination with sequence numbers to build a secure channel for transporting application-layer data. Other major components are the TLS Handshake Protocol, which is responsible for authentication, session key establishment and cipher suite negotiation, and the TLS Alert Protocol, which carries error messages and management traffic. There is also the Change Cipher Spec Protocol, which is used to signal transitioning to a protected mode. In the next sections, we describe these four protocols with more emphasis on the Record Protocol. The following are (D)TLS-related definitions that we will be regularly referring to in the course of this chapter and the rest of this thesis:

Definition 3.4. TLS cipher suite: A set of cryptographic algorithms that is agreed-on between two TLS parties. This set identifies the authentication, key establishment, bulk encryption and message authentication algorithms.

Every cipher suite is assigned a 2-byte identification number. For example, the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` is assigned number `0x2F`, and indicates the use of RSA for *authentication* and *key establishment*, AES using a 128-bit key in CBC-mode for *encryption*, and HMAC-SHA-1 for generating *message authentication codes*. IETF RFCs (for example, RFCs 5288 [90] and 5932 [52]) are published to update the list of approved and supported cipher suites for the different TLS versions [31, 32, 33]; new approved cipher suites are introduced, while existing cipher suites that are considered insecure are dropped. TLS 1.2 introduced the option of using authenticated encryption with additional data (AEAD) encryption modes [62] such as GCM [64, 90]. In AEAD encryption, the plaintext is simultaneously encrypted and integrity protected. AEAD ciphers take as input a single key, a nonce, a plaintext, and additional data to be included in the authentication check [62].

Definition 3.5. TLS session: A virtual construct that maps to a specific cipher suite and a specific active `master_secret`. Every session is assigned a random session identifier, which is an arbitrary sequence of bytes. The length of the session identifier is 16 bytes in the case of SSL 2.0 and between 0 and 32 bytes in the case of SSL 3.0 and all versions of TLS.

In Section 3.4.2, we discuss the key derivation process in (D)TLS and show how `master_secret` is computed.

Definition 3.6. TLS connection: A specific communication channel that carries the actual application data and maps to a TCP connection.

Every TLS connection is assigned its own encryption and MAC keys, and sequence numbers. A TLS connection is created only after the successful establishment of a session. The two ends of a TLS connection must agree on the same cipher suite to be able to communicate. The cipher suite to use is negotiated by the two TLS parties as described in Section 3.4.3.

3.4.1 Pseudo Random Functions in TLS

In addition to compressing data using HMAC, for integrity and authenticity purposes, another construction is required to do *expansion* of secrets into blocks of data for the purposes of key generation or validation. This is achieved by using a pseudo random function (PRF), which takes in the case of TLS as input a secret, a seed and an identifying label, and produces an output of arbitrary length. The PRF, as we show in the coming sections, takes a key role in the TLS key derivation and the Handshake Protocol; TLS relies on HMAC-based PRFs to perform key derivation, and the generation and verification of particular messages. Unlike TLS 1.0 and TLS 1.1 which hard code the HMAC algorithms to use, TLS 1.2 specifies that the PRF must be explicitly identified in the cipher suite and that it should be at least SHA-256. In TLS 1.0 and TLS 1.1, the PRF output is created by splitting the secret, S, into two halves, $S1$ and $S2$, and using $S1$ with MD5 and $S2$ with SHA-1, then performing an XOR operation on the outputs of these two expansion functions. The PRF calculation in TLS 1.2 does not require to split S; only one hash function is used.

3.4.2 TLS Key Derivation

The TLS key derivation process produces keys that are used for encryption, integrity protection and authentication. This is achieved first by creating (or computing) and then expanding a 48-byte pre-master secret, `pre_master_secret`. The process of establishing the pre-master secret depends on the key establishment method identified in the selected cipher suite. The pre-master secret is set either by direct transmission of the RSA-encrypted secret, as described in Section 3.4.3, or by the transmission of Diffie-Hellman (DH) parameters that will allow each side to reach the same `pre_master_secret` [33]. The length of `pre_master_secret` varies depending on the key establishment method. For example, in the case of RSA, `pre_master_secret` is 48 bytes long and is composed of a 2-byte version number and a 46-byte random number that is locally generated by the client and communicated “securely” to the server, using RSA encryption under the server’s public key.

The pre-master secret is then expanded using a PRF to generate a 48-byte master secret, `master_secret`. The master secret is specific to a TLS session and is maintained for the lifetime of a particular TLS session. In TLS, `master_secret` is computed by applying a PRF on the following inputs [33]: `pre_master_secret`, the string "master secret" and `ClientHello.random` || `ServerHello.random` (random values exchanged in the Handshake Protocol, as we describe in Section 3.4.3). The master secret, `master_secret`, is associated with a TLS session and is used to generate the required keying material for TLS connections under a session. The exact keying material to generate depends on the selected cipher suite and can include encryption keys, MAC keys and IVs for each of the TLS parties.

3.4.3 The TLS Handshake Protocol

Before application layer data can be exchanged between two TLS parties, a TLS connection establishment phase must complete successfully. This phase is delivered by the TLS Handshake Protocol. Figure 3.5 shows a typical sequence of the TLS Handshake messages when RSA, a public-key encryption algorithm, is used for authentication and key establishment. This Handshake Protocol example, which we explain in detail in this section, also assumes that server authentication is performed using a certificate that the client trusts; client authentication, on the other hand, is optional.

The Handshake Protocol example proceeds as follows (again, we assume that RSA is used for key establishment):

The client usually initiates the TLS Handshake Protocol. It does this by sending a `ClientHello` message, after successfully establishing the underlying TCP connection of course. Sending a `ClientHello` message indicates to the server the client's interest in starting a TLS handshake message exchange and informs the server about the client's preferences. The `ClientHello` message includes the following fields in sequence: a 2-byte protocol number (for example, `0x3 0x2` for TLS 1.1 and `0x3 0x3` for TLS 1.2), a 32-byte number referred to as `ClientRandom` (28 bytes of which are to be randomly generated), the session identifier, a variable number of bytes (multiple of two) containing the list of cipher suites supported by the client for this connection, and the compression methods the client supports (if any). All this information is sent in plaintext. It is worth noting that TLS 1.2 [33] introduced a new handshake message, `HelloRequest`, a notification message that the server can send requesting the client to begin the handshake negotiation process.

The server responds with three TLS messages, transmitted in one or more TCP datagrams. The first message is the `ServerHello` message, sent in plaintext in response to the preferences offered by the client in its `ClientHello` message. The server can ignore the client's preferences and choose another cipher suite. The `ServerHello`

```
  Client                                     Server

ClientHello              -------->
                                            ServerHello
                                           Certificate
                                 CertificateRequest(optional)
                         <--------      ServerHelloDone
Certificate(optional)
ClientKeyExchange
CertificateVerify(optional)
[ChangeCipherSpec]
Finished                 -------->
                                      [ChangeCipherSpec]
                         <--------             Finished

Application Data         <------->     Application Data
```

Figure 3.5: Sample TLS Handshake Protocol sequence of messages. The figure shows the messages exchanged between the TLS client and the TLS server in the scenario discussed in Section 3.4.3. In this scenario, we assume that RSA is used for key establishment. In this scenario, the `pre_master_secret`, which is a 48-byte key, is generated by the client and shared, securely, with the server in the third step in the `ClientKeyExchange` message. The master secret, `master_secret`, that gets associated with a TLS session and is used to generate the required keying material for the TLS connection, is computed from `pre_master_secret`. The keying material that is generated will include encryption and MAC keys used by the clinet and the server for a particular TLS connection.

message includes the following fields in sequence: a 2-byte protocol number, a 32-byte number referred to as `ServerRandom` (28 bytes of which are to be randomly generated), a session identifier assigned by the server, the 2-byte cipher suite identifier which the server is willing to use, and the compression method (if any). The `ServerRandom` and `ClientRandom` messages are used by the client and server in the key derivation process described in Section 3.4.2 to make sure that the generated keying material is different, even if the same secret input (the pre-master secret) has been chosen, protecting TLS connections from replay attacks and making sure that keys in each connection are different. In addition to `ServerHello`, the server sends a `Certificate` message containing a chain of X.509 certificates, and which must at least contain the server's certificate. The server finally sends the `ServerHelloDone` message, an empty message that indicates to the client that the server has finished sending all of its messages for this phase of the handshake.

The client continues with the Handshake Protocol only if it is satisfied with the server's response. That is, if the client accepts the server's chosen cipher suite and successfully authenticates the server's certificate it just received. If it does, then the client locally generates a secret key referred to as the pre-master secret and sends it to the server in the `ClientKeyExchange` message. The generation of the pre-master secret was described in Section 3.4.2. The `ClientKeyExchange` message is protected using the server's public key contained in the server's certificate. The `ClientKeyExchange` message is followed by the 1-byte `ChangeCipherSpec` message. This message is actually part of the TLS Change Cipher Spec Protocol and indicates that a TLS party, in this case the client, is switching to sending and accepting only messages that are protected by the selected cipher suite and the newly derived keys. The client finally sends the `ClientFinished` message, the first message that is protected by the newly derived keys used in accordance with the selected cipher suite. The `ClientFinished` message contains the client's `Verify_Data`, a cryptographically generated message used to verify that the current key exchange was successful. We show how to compute `Verify_Data` later in this section.

The server proceeds only if it successfully decrypts the `ClientFinished` message and verifies the client's `Verify_Data`. If so, then the server sends a `ChangeCipherSpec` message, followed by a `ServerFinished` message containing the server's `Verify_Data`. The client is expected to decrypt and verify the content of `ServerFinished`, before any application data is sent.

The `Verify_Data` message is cryptographically generated. For example, in TLS 1.2 [33], `Verify_Data` is computed by applying the ciphersuite-specified PRF (or SHA-256 if not specified) on the following data [33]: `master_secret`, `finished_label` and h(`handshake_messages`). The `finished_label` is set to the strings "client finished" or "server finished", if the `Verify_Data` message is generated by the client or server respectively. In TLS 1.2, the length of `Verify_Data`, in bytes, should be explicitly identified by the selected cipher suite, otherwise it is set to 12 bytes. In TLS 1.0 and 1.1, the length of `Verify_Data` is fixed to 12 bytes. Here, h is the cipher-suite-specified cryptographic hash function for the PRF and `handshake_messages` refers to all handshake messages sent or received, starting at `client_hello` and up to, but not including, the `Finished` message.

The above Handshake Protocol example is for the case in which RSA (TLS–RSA) is used for key establishment. The Handshake Protocol proceeds differently when using other key establishment and authentication algorithms. Recall that the key establishment and authentication algorithms are specified in the selected cipher suite. TLS and DTLS support cipher suites where static DH (TLS–DH) or ephemeral DH (TLS–

DHE) can be used. The DH parameters are specified by the server and may be either ephemeral (in the case of TLS–DHE) or contained within the server's certificate (in the case of TLS–DH). In TLS-DH and TLS-DHE, the pre-master secret is created from the output of the DH key exchange protocol. According to [33], the shared DH key is used as the pre-master secret, after stripping the leading bytes of the shared key that contain all zero bits. If needed, TLS–DHE can be used to achieve perfect forward secrecy (PFS), a feature that cannot be guaranteed by RSA or static DH. TLS-DHE does not provide authentication, but can be combined with an authentication method such as RSA or Digital Signature Algorithm (DSA) [33]. For example, `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` identifies DHE as the key exchange algorithm and RSA as the authentication algorithm. Using TLS with Anonymous DH, on the other hand, is possible, but does not provide authentication; it is strongly recommended not to use anonymous DH since it is vulnerable to basic MITM attacks.

3.4.4 TLS Session Renegotiation

Two TLS parties can renegotiate the parameters of their session without the need to tear down their TLS connection. Re-negotiated parameters can include the cipher suite and the pre-master secret, along with recomputing the master secret accordingly. The session renegotiation capability is negotiated when two TLS parties first establish a TLS session. Unlike most of the messages of a new handshake which are sent in plaintext, all renegotiation messages are protected by the current cipher suite. RFC 5746 [84] defines a secure renegotiation method, published in response to a TLS renegotiation attack, described in [81].

3.4.5 TLS Session Resumption

To avoid the computationally expensive public key cryptography operations used to establish the pre-master secret, a client and a server can *resume* a previously established session. Successful session resumption results in updating the TLS keying material and possibly a change in the cipher suite; the master secret associated with the TLS session is not changed. A non-empty session identifier in an unencrypted `ClientHello` indicates to the server the client's interest to resume an existing session. The session resumption mechanism is also referred to as "session caching", and requires the server to keep information about every session for a configurable amount of time. The authors of RFC 5077 [92] introduces a new TLS extension for stateless session resumption that does not make use of session identifiers.

3.4.6 The TLS Record Protocol

The TLS Record Protocol transparently provide services the TLS higher-level protocols. The Record Protocol accepts data from its higher-level protocols and from the underlying reliable transport protocol layer (for example, TCP). Data received from higher-level protocols are first fragmented into records, compressed (if selected) and encapsulated in TLS datagrams. Application messages exceeding 2^{14} bytes are broken into multiple plaintext records, each not exceeding 2^{14} bytes, before they are further processed. The Record Protocol applies authentication, padding and encryption to each plaintext record, as appropriate. The exact list of steps for processing TLS plaintext records is governed by the state of the TLS connection and the cipher suite selected during the TLS connection establishment phase. For example, some of the TLS handshake messages are sent in plaintext, while others are protected by the selected cipher. Data received from applications are always protected by the selected cipher. According to [33], the length of a TLS ciphertext record must not exceed $2^{14}+2^{12}$ bytes. The TLS Record Protocol performs the reverse operations on data received from its underlying reliable transport protocol. Again, the exact processing of TLS records is governed by the state of the TLS connection and the selected cipher suite.

Let us now analyse the structure of the TLS record header. The 5-byte header, `HDR`, is simple and consists of three fields: `Content Type`, `Version` and `Length`. The 1-byte `Content Type` field contains the value assigned to the higher-level protocol. The four basic higher-level protocols are the Change Cipher Spec Protocol (`0x14`), the Alert Protocol (`0x15`), the Handshake Protocol (`0x16`), and the Application Data Protocol (`0x17`). The Record Protocol supports adding extensions so that new higher-level protocols can be introduced whenever needed [33, 2]. The 2-byte `Version` field, identifies the version of the protocol used, and is broken into 1-byte `Major` and 1-byte `Minor` versions. For example, TLS 1.2 is assigned the value `0x3 0x3`. The 2-byte `Length` field identifies the length of the TLS record in bytes.

Sequence numbers are used by the Record Protocol to maintain the state of a TLS connection. The value of an 8-byte sequence number, `SQN`, is initialised to zero whenever a new connection is established. The 8-byte sequence number is not exchanged over the wire, but is maintained by the two parties of a TLS connection and is protected by the Record Protocol MAC. Two separate sequence numbers are maintained for every TLS connection; one for outgoing records and another for incoming records. The corresponding sequence number is incremented for each record sent or received. Sequence numbers are also used to protect TLS against anti-replay attacks. Although TCP provides a reliable layer that assures in-sequence delivery of datagrams, it does not protect TLS against adversaries intentionally re-ordering TCP datagrams or injecting

TCP datagrams of their choice. The TLS Record Protocol detects this manipulation or injection by continuously maintaining the value SQN and including SQN in the MAC calculation. The MAC is verified for every TLS record received and the TLS session is destroyed immediately, along with its associated keying material, after encountering an invalid MAC.

3.4.7 Modes of Operation

In the Record Protocol, there are three encryption options:

- HMAC followed by CBC-mode encryption using a block cipher,
- HMAC followed by encryption using the RC4 stream cipher, or
- authenticated encryption using the GCM [90] or CCM [63] mode of operation of a block cipher. In this mode of operation, the same construction is used to perform the expected encryption and authentication functions.

The third of these three options is only available with TLS 1.2 [90, 63], which is yet to see widespread adoption. The second option has seen some recent cryptanalysis work [4]. Our attacks, presented in Chapters 4 and 5, were conducted against DTLS and TLS when CBC-mode encryption is used. Therefore, the background information we give in the remaining of this chapter focuses on the TLS Record Protocol with this mode of operation, which we refer to as MEE-TLS-CBC in the rest of the thesis. In this case, MEE-TLS-CBC refers to using the MAC-then-Encode-then-Encrypt (MEE) construction or TLS (and DTLS), under CBC-mode encryption.

CBC-Mode Encryption in TLS – MEE-TLS-CBC

An individual TLS plaintext record R (viewed as a byte sequence of length at least zero) is processed as follows: the sender maintains the 8-byte sequence number SQN, and forms the 5-byte header field HDR described earlier. It then calculates a MAC over the bytes $\texttt{SQN} \,||\, \texttt{HDR} \,||\, R$; let T denote the resulting MAC tag. Note that exactly 13 bytes of data are prepended to the record R here before the MAC is computed. The size of the MAC tag is 16 bytes (HMAC-MD5), 20 bytes (HMAC-SHA-1), or 32 bytes (HMAC-SHA-256). We let t denote this size in bytes.

The record is then encoded to create the plaintext P by setting $P = R \,||\, T \,||\, \texttt{pad}$. Here pad is a sequence of padding bytes chosen such that the length of P in bytes is a multiple of b, where b is the block-size of the selected block cipher (so $b = 8$ for 3DES and $b = 16$ for AES). In all versions of TLS and DTLS, the padding must consist of

$p+1$ copies of some byte value p, where $0 \leq p \leq 255$. In particular, at least one byte of padding must always be added. So examples of valid byte sequences for `pad` are: "0x00", "0x01 || 0x01" and "0x02 || 0x02 || 0x02". The padding may extend over multiple blocks, and receivers must support the removal of such extended padding. In SSL the padding format is not so strictly specified: it is only required that the last byte of padding must indicate the total number of additional padding bytes. However, this opens up TLS to simple attacks as described in [68].

In the encryption step, the encoded record P is encrypted using CBC-mode of the selected block cipher. TLS 1.1 and 1.2 mandate an explicit IV, which should be randomly generated; TLS 1.0 and SSL use a chained IV. Thus, the ciphertext blocks are computed as:

$$C_j = E_{K_e}(P_j \oplus C_{j-1}), \tag{3.1}$$

where P_j are the blocks of P, C_0 is the IV, and K_e is the key for the block cipher E, that was agreed-on during the TLS handshake phase.

For TLS (and SSL), the data transmitted over the wire then has the form:

$$\texttt{HDR} \,||\, C,$$

where C is the concatenation of the ciphertext blocks C_i (including or excluding the IV depending on the particular SSL or TLS version). Note that the sequence number is not transmitted as part of the message in TLS.

Simplistically, the decryption process, shown in Figure 3.6, reverses this sequence of steps: first the ciphertext is decrypted block-by-block to recover the plaintext blocks:

$$P_j = D_{K_e}(C_j) \oplus C_{j-1}, \tag{3.2}$$

where D denotes the decryption algorithm of the block cipher. Then the padding is removed, and finally, the MAC is checked, using the header information (and, in TLS, a version of the sequence number that is maintained at the receiver).

In reality, much more sophisticated processing than this is needed. We describe the further steps to perform later in Section 3.9.2.

3.4.8 The TLS Alert Protocol

The TLS Alert Protocol [33] carries alert messages which identify the severity of errors occurring at any stage of a TLS connection. For example, an alert message is sent by a server if none of the cipher suites offered by the client, during the Handshake Protocol, are acceptable. Alert messages are classified as either fatal or warning. Alert messages

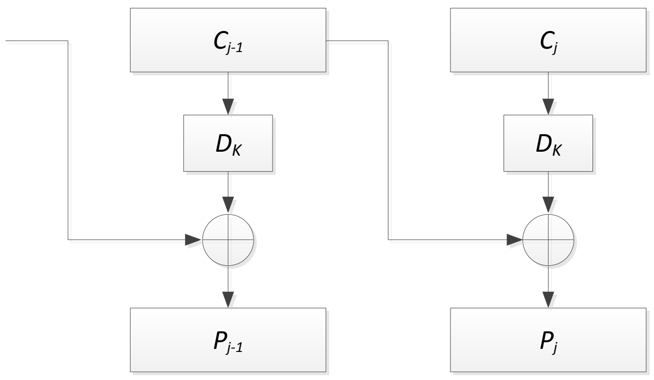


Figure 3.6: CBC-mode decryption. In this figure, C is the concatenation of the ciphertext blocks C_i (including or excluding the IV depending on the particular SSL or TLS version), while P is the decrypted ciphertext and K is the key used for decryption.

with a level of fatal result in the immediate termination of the TLS connection and destruction of the session construct accordingly.

3.5 Introduction to Datagram Transport Layer Security

A datagram-based protocol implies the use of an unreliable underlying network protocol such as UDP. The Datagram Transport Layer Security (DTLS) protocol was first introduced at the Network and Distributed System Security Symposium (NDSS) in 2004 [67]. Two years later, the IETF assigned RFC 4347 [82] to DTLS 1.0. The aim of DTLS 1.0 is to provide a variant of TLS 1.1 [32] that eliminates the dependency on a reliable underlying protocol such as TCP. Similar to TLS, DTLS is intended to provide privacy and integrity for data exchanged between a client and a server. We explain later in this section the main differences between DTLS and TLS. Applications that operate over an unreliable transport protocol such as UDP can easily take advantage of the security services offered by DTLS. Since its introduction, there has been a growing interest in the security services offered by DTLS. Leading implementations of DTLS can be found in OpenSSL[2] and GnuTLS[3]. Both of these provide source toolkits that implement TLS and DTLS as well as being general purpose cryptographic libraries that software developers can use. The first release of OpenSSL to implement DTLS was 0.9.8. Since its release, DTLS has become a mainstream protocol in OpenSSL. There are also a number of commercial products that have taken advantage of DTLS.

[2]http://www.openssl.org
[3]http://www.gnu.org/software/gnutls

For example, DTLS is used to secure Virtual Private Networks (VPNs)[4,5] and wireless traffic[6]. Platforms such as Microsoft Windows, Microsoft .NET and Linux can also make use of DTLS[7]. In addition, the number of RFC documents that are being published on DTLS is increasing. Recent examples include RFC 5415 [24], RFC 5953 [44] and RFC 6012 [91]. Recently, support for DTLS was added to Google Chrome and Firefox to protect Internet video traffic[8].

By design, DTLS 1.0 [82] is very similar to TLS 1.1 [32]. In fact, RFC 4347 [82] presents only the changes to TLS 1.1 introduced by DTLS and refers to RFC 4346 [32] for the rest of the protocol specification. According to RFC 4347, this approach has been chosen to minimise the amount of effort needed to implement the protocol. Thus, to fully understand and be able to analyse and code DTLS, the reader of RFC 4347 is expected to be familiar with TLS 1.1. The same approach was taken when developing DTLS 1.2 [83], which aligns the DTLS protocol with TLS 1.2. It is worth noting that there was no DTLS 1.1. The jump was to synchronise the DTLS and TLS numbering for ease of referral, and possibly development and implementation.

A number of changes were introduced in the design of DTLS, compared to TLS, so that the services of TLS could be delivered over an unreliable transport protocol such as UDP. We list here a number of these changes:

- To compensate for the lack of an underlying reliable protocol, the DTLS Handshake Protocol implements its own retransmission timers and datagram reordering settings. The Handshake Protocol also implements other features such as anti-spoofing and denial of service protection. For example, to protect against denial of service attacks, DTLS introduces a handshake verification phase in which a challenge message, `HelloVerifyRequest`, in the form of a 4-byte cookie, borrowed from [51], is generated by the server and sent to the client. The client is expected to retransmit the `ClientHello`, this time with the cookie included, making denial of service using spoofed IP addresses difficult. Figure 3.7 shows the new handshake phases. The rest of the DTLS handshake proceeds similarly to the TLS handshake example shown in Figure 3.5, along with the implementation of message timeouts. The Record Protocol does not offer these features and assumes that they are handled by the upper layer protocols, if needed.

- In TLS, MAC errors must result in connection termination. In DTLS, the receiv-

[4] `http://www.cisco.com/en/US/products/ps10884/index.html`
[5] `http://campagnol.sourceforge.net`
[6] `http://www.cisco.com/en/US/docs/wireless/controller/7.0MR1/configuration/guide/cgi_lwap.html`
[7] `http://www.eldos.com/sbb/desc-ssl.php`
[8] `http://sites.google.com/site/webrtc/interop`

ing implementation may simply discard the offending record and continue with the connection. According to [83, Section 4.1.2.1], DTLS implementations should silently discard data with bad MACs. The idea is that datagrams may be lost, duplicated, reordered, or possibly modified. Discarding these packets is based on the fact the underlying protocol used by DTLS is unreliable. TLS, on the other hand, would terminate the connection since it expects the underlying protocol, for example TCP, to carry out the functions of packets re-ordering and discarding packets with checksum errors. We exploit the DTLS tolerance to datagram alteration in our attacks in Chapter 4.

- Unlike TLS, fragmentation of record messages is not permitted in DTLS. Instead, a DTLS record must fit within a single lower layer datagram. According to [83, Section 4.1.1], DTLS implementations are expected to determine the path maximum transmission unit (PMTU) possible and send records smaller than the PMTU. If an application attempts to send a record larger than the allowed PMTU, the DTLS implementation should respond to the application with an error message.

- Unlike TLS, the 8-byte sequence number field, `SQN`, in DTLS is explicit, i.e. is included in the DTLS Record header, and is composed from a 2-byte *epoch* number and a 6-byte sequence number, As with TLS, the 6-byte DTLS sequence number is set to zero after each `ChangeCipherSpec` message is sent. The epoch number is initially set to zero and incremented each time the `ChangeCipherSpec` message is sent within a session (for reasons such as session renegotiation). For simplicity, we use DTLS sequence number field, `SQN`, to refer to both the 2-byte epoch and the 6-byte sequence number.

- DTLS optionally supports record replay detection. TLS support for anti-replay is based on including the current `SQN` in the MAC calculation as described in Section 3.4.6. In DTLS, the technique used for anti-replay is the same as in IPsec's AH protocol [53], by maintaining a bitmap window of received records. Records that are too old to fit in the window and records that have previously been received are silently discarded. According to [82, 83], the replay detection feature is optional, since packet duplication is not always malicious, but can also occur due to routing errors. In DTLS, the 8-byte sequence number field, `SQN`, is included in the DTLS Record header; this value is used for calculating and verifying the MAC.

- RC4, the only stream cipher that is supported by TLS, must *not* be used with

```
   Client                                Server
   ------                                ------

ClientHello           -------->

                      <-------    HelloVerifyRequest

ClientHello           -------->
```

Figure 3.7: Startup of DTLS Handshake Protocol.

DTLS. This is because the DTLS Record Protocol and its underlying transport layer protocol do not provide reliability, message delivery assurance or datagram in-order processing, and hence the state synchronisation required by stream ciphers like RC4 cannot be guaranteed.

3.6 Heartbeat Extension for (D)TLS

The Heartbeat extension [93] provides a new protocol for (D)TLS allowing a keep-alive functionality. This is very useful in the case of DTLS, which runs on top of unreliable transport protocols that have no concept of session management. The only mechanism available at the (D)TLS layer to determine if a peer is still alive is performing a costly renegotiation. The Heartbeat extension uses Heartbeat request and response messages between two entities that have an established (D)TLS connection. A Heartbeat request message can be sent by either of the entities and is protected using the same (D)TLS cipher suite and keys used for protecting other payloads. According to [93], whenever a Heartbeat request message is received, it has to be answered with a corresponding Heartbeat response message. Both messages have specific lengths that can be detected by the adversary. The use of (D)TLS's variable length padding feature adds minimal difficulty in identifying the Heartbeat messages. We make use of the (D)TLS Heartbeat messages in our attacks in Chapters 4 and 5.

3.7 Implementations of (D)TLS

TLS 1.0 is universally supported by almost all modern web browsers and web servers. TLS 1.1 and 1.2 are not yet widely supported[9,10], but TLS 1.2 is gaining further momentum, after a number of recent high-profile attacks against TLS [35, 4] and Chapter 5 of this thesis (published as [6]). Examples of widely-used open source implementations of TLS 1.2 include OpenSSL[11], NSS[12] (used in Google Chrome and Firefox), GnuTLS[13], PolarSSL[14], JSSE[15], CyaSSL[16] and yaSSL[17]. Some vendors such as Microsoft have opted to develop their own implementations of the protocol[18]. The number of DTLS 1.1 implementations is fewer and includes, for example, OpenSSL, GnuTLS and CyaSSL.

The implementers of TLS and DTLS are expected to follow the IETF standards for their code design and implementation. Despite their best efforts, vulnerabilities can be introduced through developers making their own interpretation of the standards or as a result of the usual kinds of coding error. There are also cases where vulnerabilities are the result of incorrect protocol design decisions. We demonstrate examples of such vulnerabilities later in Chapters 4 and 5, and introduce new techniques for exploiting (D)TLS design decisions and attacking a number of (D)TLS implementations.

3.8 Side Channel Attacks

Physical attacks on cryptosystems take advantage of implementation-specific characteristics to recover secret parameters that are involved in different computations. An attack that exploits information leaked by a system is generally referred to as a side channel attack; typically, exploiting *unintended* leakage that hardware and software implementations of cryptosystems may produce. Sources of hardware leakage include timing [55, 23], power consumption [54] and electromagnetic radiation [56]. Sources of software leakage include error messages [73, 30, 102, 16] and message sizes [8]. An

[9]SSL Pulse (https://www.trustworthyinternet.org/ssl-pulse/) reported in August 2013 that only 14.5% of 170,000 websites surveyed support TLS 1.1 and 17% of the of 170,000 websites support TLS 1.2.

[10]As of September 2013, Firefox support for TLS 1.2 is available on selected releases (Firefox 24 Beta, Aurora and Nightly), Google Chrome support for TLS 1.2 was introduced in release 29, and Microsoft plans to support TLS 1.2 in release 11 of its Internet Explorer.

[11]http://www.openssl.org

[12]http://developer.mozilla.org/en-US/docs/NSS

[13]http://www.gnutls.org

[14]http://polarssl.org

[15]http://download.java.net/jdk8/docs/technotes/guides/security/jsse/JSSERefGuide.html

[16]http://www.yassl.com

[17]http://www.yassl.com

[18]http://msdn.microsoft.com/en-us/library/windows/desktop/aa380123

attacker exploiting a side channel can gain further information that may potentially help him exploit a cryptosystem. For example, this might help an attacker recover the encryption key or recover encrypted messages.

3.8.1 Timing Side Channel Attacks

A timing side channel attack is, essentially, a way of obtaining secret information by measuring the time it takes for cryptographic operations to complete. The timing variance in the operations reveals enough information to the attacker for him to acquire relevant, and possibly sensitive, information about the system. Timing side channel attacks are typically coupled with statistical analysis. The timing side channel attack idea was first introduced by Kocher in [55], where he showed that carefully measuring the amount of time required to perform private key operations could possibly help attackers find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. The fundamental idea is taking advantage of timing variations. The idea was then translated into a real attack against a smart card-based implementation of RSA [54]. Boneh and Brumley later took this further and demonstrated in [23] how to extract private keys from an OpenSSL-based web server running on a machine in the local network. The work we present in Chapters 4 and 5 exploits timing side channels to recover plaintext.

3.9 Padding Oracles

An oracle can be thought of as a black box that responds to queries. A padding oracle reveals side channel information that indicates the correctness of padding. In certain circumstances, a padding oracle can be leveraged to build a decryption oracle, that is, enables plaintext attacks against a protocol. For it to be useful to an adversary, a padding oracle must have a practical realisation. This realisation can be achieved by exploiting the leakage of padding-related side channel information such as error messages or timing.

The concept of a padding oracle was first introduced by Vaudenay [102]. In Vaudenay's formulation, a padding oracle is a notional algorithm which, when presented with a CBC-mode ciphertext, returns `VALID` if the underlying plaintext has padding that is correctly formatted and `INVALID` otherwise. Here, correctness is with respect to some padding scheme. For example, for (D)TLS padding, correctness means that the decryption of the ciphertext is a byte string ending in one of the valid padding patterns "`0x00`", "`0x01 0x01`", etc. Vaudenay showed that, for certain padding schemes, repeated access to a padding oracle can be used to decrypt arbitrary target ciphertext blocks (and indeed complete ciphertexts in a block-by-block manner).

Vaudenay's techniques apply to the (D)TLS padding scheme and, for completeness, we show in Algorithm 2 how to decrypt a complete block from a target ciphertext C, given access to a padding oracle. Recall that in CBC-mode encryption, the ciphertext is decrypted block-by-block using equation (3.2), demonstrated by Figure 3.6.

Let us start with the simplest scenario, in which the attacker tries to recover the last byte of a plaintext block, P_t. We use C_t to denote the target block in ciphertext C and use C_{t-1} to denote the ciphertext block preceding C_t. Let Δ, which we refer to as the *masking* block, be a block made of b bytes. Also, for any block B of plaintext or ciphertext, we write $B = [B[0]B[1] \dots B[b-1]]$, where $B[i]$ denotes the ith bytes of B.

In CBC-mode encryption, modifying the value of a byte of C_{t-1} by XORing it with Δ has the effect of modifying P_t in the same byte by XORing it with the same Δ, as shown in Figure 3.8. In addition, modifying any byte in C_{t-1} would result in unpredictable changes to plaintext block P_{t-1}; this is due to the characteristics of the block cipher used. It is worth highlighting at this point that any change in C would, with overwhelming probability, invalidate the (D)TLS record's MAC. In this text, we refer to the modified versions of C and P as C^* and P^* respectively. If the attacker can place C_{t-1} and C_t as the last two ciphertext blocks, then a change in $C_{t-1}[b-1]$ (the last byte of C_{t-1}) would result in a change in $P_t[b-1]$, which is treated in (D)TLS as the padding length byte. Recall that in all versions of TLS and DTLS, the padding must consist of $p+1$ copies of some byte value p, where $0 \leq p \leq 255$. Here:

$$P^*_t[b-1] = \Delta[b-1] \oplus P_t[b-1]. \tag{3.3}$$

The attack proceeds as follows. The attacker initialises $\Delta[b-1]$ to 0 and constructs C^* using:

$$C^* = C_{t-1}[0]C_{t-1}[1] \dots (C_{t-1}[b-1] \oplus \Delta[b-1]) \mid\mid C_t. \tag{3.4}$$

The attacker then submits C^* to the padding oracle, $\mathcal{PO}$. If $\mathcal{PO}$ responds with `INVALID`, then the attacker increments the value of $\Delta[b-1]$, constructs a new C^* (using the above formula) and submits the new C^* to $\mathcal{PO}$. The attacker stops when $\mathcal{PO}$ responds with `VALID`, indicating valid padding ("`0x00`" in this case – note that the probability of encountering the other valid padding combinations than "`0x00`" is considerably lower). The attacker recovers $P_t[b-1]$ using:

$$\begin{aligned} P_t[b-1] &= \Delta[b-1] \oplus \texttt{0x00} \\ &= \Delta[b-1]. \end{aligned} \tag{3.5}$$

The attack takes on average 128 and at most 256 queries to $\mathcal{PO}$ to recover $P_t[b-1]$.

So far, we described how to recover only the last byte of a block. Algorithm 2 describes how to decrypt a complete ciphertext block using $\mathcal{PO}$. The attack as presented in Algorithm 2 uses 2-block ciphertexts, but is easily adapted to use longer ciphertexts simply by ensuring that blocks C^*_{t-1}, C_t are always placed at the end of the ciphertext. The fundamental idea is that the attacker recovers a block, one byte at a time, starting from the *rightmost* byte, $P_t[b-1]$. When trying to recover a plaintext byte $P_t[i]$, $\Delta[i]$ is initialised to 0, where $0 \leq i < b$. In addition, all plaintext bytes in positions $i+1$ to $b-1$, if any, and which have been already recovered by now, are set to $b-i-1$ by modifying the corresponding bytes in Δ using:

$$\Delta[j] = P_t[j] \oplus (b-i-1), \tag{3.6}$$

for $i < j < b$. The attacker computes $C^*_{t-1}[b-i-1]$ using:

$$C^*_{t-1}[b-i-1] = C_{t-1}[b-i-1] \oplus \Delta[b-i-1]. \tag{3.7}$$

Starting with $\Delta[b-i-1] = 0$, the attacker computes $C^*_{t-1}[b-i-1]$ and submits $C^* = C^*_{t-1} \mid\mid C_t$ to $\mathcal{PO}$. A `VALID` response from $\mathcal{PO}$ for some value of $\Delta[b-i-1]$, indicates valid padding, i.e. $P^*_t[b-i-1] = i$ for $0 \leq i < b$. The attacker can now

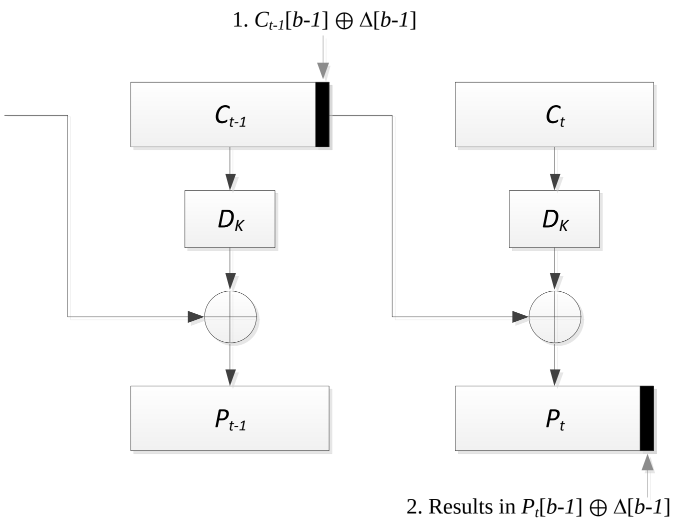

1. $C_{t-1}[b-1] \oplus \Delta[b-1]$
C_{t-1}
C_t
D_K
D_K
P_{t-1}
P_t
2. Results in $P_t[b-1] \oplus \Delta[b-1]$

Figure 3.8: In this example, where CBC-mode encryption is used, $C_{t-1}[b-1] \oplus \Delta[b-1]$ has the same effect of $P_t[b-1] \oplus \Delta[b-1]$. The other bytes of P_t are not affected. In CBC-mode encryption, modifying the value of a byte of C_{t-1} by XORing it with Δ has the effect of modifying P_t in the same byte by XORing it with the same Δ.

Algorithm 2: Decrypting a block using a padding oracle, $\mathcal{PO}$, with block-mode encryption.

```
input  : $C_{t-1}, C_t$
output: $P_t$

Initialise all bytes of $\Delta$ to 0;
for $i = 0$ to $b - 1$ do
    for byte $= 0$ to 255 do
        $\Delta[b - i - 1] \leftarrow$ byte;
        $C^* = (C_{t-1} \oplus \Delta) \,||\, C_t$;
        if $\mathcal{PO}(C^*) =$ VALID then
            $P_t[b - i - 1] = \Delta[b - i - 1] \oplus i$;
            Break;
    for $j = 0$ to $i$ do
        $\Delta[b - j - 1] = P_t[b - j - 1] \oplus i$;
Output $P_t$;
```

compute $P_t[b - i - 1]$ using:

$$P_t[b - i - 1] = \Delta[b - i - 1] \oplus i. \tag{3.8}$$

The attack requires on average 128 and at most 256 queries to the padding oracle to decrypt a byte. In the case of TLS, things are not as simple; there are complications to consider when building a padding oracle realisation for $\mathcal{PO}$. We discuss these complications in the next section.

3.9.1 Using a Padding Oracle to Attack TLS

In practice, to mount a padding oracle attack, an adversary must find some way of actually realising a padding oracle for a specific implementation. In the original presentation for TLS in [102], Vaudenay posited that such an oracle could be built by sending a message to a TLS server and then waiting for a replay in the form of an error message. In TLS 1.0, a `decryption_failed` message would indicate a padding error, while a `bad_record_mac` message would indicate that padding was correct, but that MAC verification had failed. There are (at least) two challenges to building a TLS padding oracle in this way:

1. The two TLS errors, `decryption_failed` and `bad_record_mac`, are classified as fatal, causing the immediate termination of the TLS connection after *every* query to the padding oracle. Informally, we say that the padding oracle behaves as a

bomb oracle. The adversary must wait for a new TLS connection to be established before making another query, but each new connection will have fresh keying material. This makes the attack impractical unless connections are re-established quickly. Moreover, unless the same plaintext is repeated in a known ciphertext block across many connections, the adversary can not efficiently recover plaintext. For example the probability of recovering the last byte of a block is $1/2^8$, while the probability of recovering the byte before the last one in a block is $1/2^{16}$.

2. The two error messages are encrypted, making it more difficult for the adversary to distinguish them.

Based on this, the attack against TLS was flagged as impractical, until the introduction of a side channel by Canvel *et al.*

3.9.2 Canvel *et al.* Timing Attack Against TLS

The work of Canvel *et al.* [25] addressed the second issue above, by developing a different realisation for the padding oracle[19].

The Canvel *et al.* realisation of the padding oracle relies on the fact that, for a TLS implementation, the processing of a message with valid padding may take longer than the processing of a message with invalid padding. The reason for this is that the padding is checked for validity before the MAC verification is performed, and so a TLS implementation that aborts processing immediately after detecting an error (of any kind) will exhibit a timing difference in message processing for packets with valid and invalid padding: in the former case, the MAC verification will take place, while in the latter it will not. The timing difference would then show up as a difference in the time at which the error messages appear on the network. As observed in [25], this is exactly how TLS was implemented in OpenSSL.

In the attack of [25], the timing difference was amplified by working with long messages, since these take longer to pass through MAC verification. Canvel *et al.* reported timing differences of as much as 2 milliseconds for these long messages. Because of noise introduced by various sources, the padding oracle so obtained is not fully reliable, so the server had to be queried a number of times for every message and a statistical model used to analyse the observed timings. Moreover, the oracle is still a bomb oracle, so only one query per TLS connection can be made. Even so, Canvel *et al.* [25] were able to use this approach to extract TLS-encrypted passwords for an IMAP e-mail server running stunnel, an application using the OpenSSL implementation of TLS. The attack

[19]The reader might find our description of the work of Canvel *et al.* to be different from the original; this is merely for the purpose of presenting the work in the context of padding oracles.

of Canvel *et al.* assumes the multi-session setting, in which the same plaintext is sent in the same position, in multiple sessions.

Countermeasures

The attack of Canvel *et al.* was perceived as serious enough that the OpenSSL code for TLS was updated from releases 0.9.6i and 0.9.7a in an *attempt* to ensure that the processing time for TLS messages is essentially the same, whether or not the padding is correct, and to send the same encrypted error message, `bad_record_mac`, in both cases. Eventually, the same countermeasures appeared in the specification for TLS 1.1 [32], with the requirement that they *must* be implemented.

Other Issues to Consider

Recall our description of CBC-mode decryption for TLS in Section 3.4.7, in which we highlighted that much more sophisticated processing is required than that discussed in the section. Here, we expand on that issue.

The receiver of a (D)TLS record should also check that the ciphertext size is a multiple of the block size and is large enough to contain at least a zero-length record, a MAC tag of the required size, and at least one byte of padding to avoid underflow conditions that could lead to denial of service or other severe attacks. After decryption, the receiver should check that the format of the padding is one of the possible patterns when removing it, otherwise attacks are possible [68] (SSL allows a loose padding format, while no specific padding checks are enforced during decryption in TLS 1.0, so both are potentially vulnerable to the attacks in [68]). Typically this is done by examining the last byte of the plaintext, treating it as a padding length byte `padlen`, and using this to dictate how many additional bytes of padding should be removed. But care is needed here, since blindly removing bytes could result in an underflow condition: there needs to be sufficient bytes in the plaintext to remove a total of `padlen+1` bytes and leave enough bytes for at least a zero-length record and a MAC tag.

If all this succeeds, then the MAC can be recomputed and compared to the MAC tag in the plaintext. If the padding fails to be correctly formatted, then implementations should continue to perform a MAC check anyway, to avoid providing the timing side-channel of the type exploited by the attack of Canvel *et al.* But since the padding format is incorrect in this case, it's not immediately clear where the padding ends and the MAC tag is located: in effect, the plaintext is now unparseable. The solution recommended in TLS 1.1 and 1.2 is to assume zero-length padding, interpret the last t bytes of the plaintext as a MAC tag, interpret the remainder as the record R and run

MAC verification on $\texttt{SQN} \,||\, \texttt{HDR} \,||\, R$.

3.10 The BEAST Attack

The so-called BEAST attack [35] is a blockwise-adaptive chosen-plaintext attack (BACPA) that can be mounted against SSL and TLS 1.0. This attack exploits the use of chained initialisation vectors (IVs) for CBC-mode and has its roots in [88, 68, 14, 15].

The BEAST attack achieved full plaintext recovery against SSL and TLS 1.0, but only in scenarios where an attacker can gain access to a chosen plaintext capability, perhaps by inducing the user to first download malicious JavaScript code into his browser. It is worth noting that the attack presented in [35] was constrained to the web setting, i.e. where TLS communications take place between a browser and a web server. In [35], it is assumed that the attacker has network eavesdropping, chosen-boundary and blockwise privileges. The chosen-boundary privilege allows the attacker to control block boundaries by prepending variable length sequences of bytes to a record, while the blockwise privilege allows him to prepend plaintext blocks to ongoing requests [35, 68, 15], i.e. the attacker can insert his block as the first block for encryption. The network eavesdropping privilege provides the attacker with access to $\mathcal{O}_{\mathcal{BA}}$, an oracle that returns `TRUE` when two ciphertext blocks match and `FALSE` otherwise. The reader will find that our description of the attack and the symbols we use slightly differ from [68, 14, 15]. Our goal is to give the reader a clear description of the attack that could be easily implemented. We first describe the attack model, list a number of assumptions and discuss how to implement the three privileges described above. We then describe the steps that the attack takes to recover an unknown message, one byte at a time.

The victim, in this case a user with a web browser, is somehow induced to visit a web site that hosts the attacker's malicious code (for example, a malicious JavaScript) which gets downloaded to his browser. By way of example, the attacker can try to inject an `iframe` tag in a website that the victim normally visits or simply send the victim an email with an embedded link pointing to the malicious web site. For simplicity, we refer to the malicious code as the web agent (`WA`). When executed, the web agent sends an HTTPS request to a web server (for example, "www.paypal.com"). The attacker's goal is to recover all or part of the TLS-protected message in the victim's HTTPS request to the web server. Web cookies [17], contained in web requests, make a good target for attackers to try to recover. The web agent HTTPS request would trigger the establishment of a TLS (or SSL) connection, in case one does not already exist. We assume that CBC-mode encryption is used.

Let us take the example in which the attacker is trying to recover an l-byte cookie.

We assume that the attacker knows l and knows the exact position of the cookie in the web request. For the attack to work, continuous communication between the web agent and a network eavesdropping agent (NA) is required. The network eavesdropping agent has access to the ciphertext blocks only and implements the blockwise-adaptive oracle, $\mathcal{O}_{\mathcal{BA}}$. In addition, it is important that the characters prepended by the attacker do not cause the server to abandon the session. The web agent must also be able to bypass the browser's same-origin policy (SOP)[20], a browser-based security mechanism that controls which messages one origin (web site) can send to another through the browser. Clearly, a good number of prerequisites must be satisfied for the attack to work.

In practice, the original web request can be sent over multiple records; for simplicity, we assume that the cookie is contained in one TLS plaintext record. The attacker tries to recover the cookie one byte at a time. Recall that the attacker can prepend a sequence of characters of his choice to the request, through the web agent, making use of the chosen-boundary privilege, always positioning the targeted *unknown* byte at the end of a block in which all other bytes are already known. The exact number of bytes to prepend, m, is adjusted by the web agent during the attack. Let us assume that the *current* plaintext record, which contains the attacker's prepended bytes, is P. P is made of blocks $P_1P_2 \ldots P_n$, where n is the total number of blocks including padding and MAC. Applying CBC-mode encryption on P generates the corresponding ciphertext blocks, $C_1C_2 \ldots C_n$.

Let us now describe how to implement the actual attack that recovers the cookie (or any other unknown message). Once the web agent is successfully downloaded, executed and able to communicate with the network eavesdropping agent, the attack proceeds in two iterative steps to recover one byte of plaintext at a time:

Step 1: The web agent sends a web request that positions the *current* unknown byte at the end of P_j, that is at position $P_j[b-1]$. It does this by prepending the correct number of bytes, m, to the web request (implementing the chosen-boundary privilege). In this case, all the bytes in P_j, except $P_j[b-1]$, are known to the web agent. The web agent submits P to the web browser which encrypts it to obtain C.

Step 2: Starting with $i = 0$, the attacker computes:

$$P_i^* = C_n^{'} \oplus C_{j-1} \oplus (P_j[0], ..., P_j[b-2], i), \tag{3.9}$$

where $0 \leq i \leq 255$, $C_n^{'}$ is the last block in the previous record, $C^{'}$, and C_{j-1} is the

[20]http://www.w3.org/Security/wiki/Same_Origin_Policy

ciphertext block preceding C_j. Recall that all the plaintext bytes of P_j, except $P_j[b-1]$, are known to the attacker. The attacker prepends P_i^* to the request, i.e. the attacker positions P_i^* as the *first* block in the record for encryption. In SSL and TLS 1.0, the last block of the previous ciphertext record, $C_n^{'}$, is used as the IV when processing the current record; the ciphertext block C_i^* is calculated as follows:

$$\begin{aligned} C_i^* &= E_{K_e}(C_n^{'} \oplus P_i^*) \\ &= E_{K_e}(C_n^{'} \oplus C_n^{'} \oplus C_{j-1} \oplus (P_j[0], ..., P_j[b-2], i)) \\ &= E_{K_e}(C_{j-1} \oplus (P_j[0], ..., P_j[b-2], i)). \end{aligned} \tag{3.10}$$

C_i^* and C_j are then submitted to $\mathcal{O}_{\mathcal{BA}}$, which is implemented by the eavesdropping agent. The network eavesdropping and the web agent continuously communicate so that i is incremented as needed. If $\mathcal{O}_{\mathcal{BA}}$ returns `TRUE`, i.e. $C_i^* = C_j$, then the attacker concludes that $i = P_j[b-1]$ as per equation (3.10). Otherwise, the attacker (the web agent in this case) increments i, computes another C_i^* and queries $\mathcal{O}_{\mathcal{BA}}$. The attack takes on average 128 and at most 256 queries to $\mathcal{O}_{\mathcal{BA}}$ to recover a byte. Once $P_j[b-1]$ is recovered, the attacker then targets the next plaintext byte in the cookie by performing **step 1** and **step 2** accordingly, until the whole cookie is recovered.

3.10.1 Countermeasures

Despite its strong requirements (the three privileges explained earlier and bypassing the SOP restriction), the BEAST attack attracted significant industry and media attention in 2011. The author of [15] suggested a number of countermeasures to defeat BACPA, well before the BEAST attack was released:

- Upgrade to TLS 1.1 (or TLS 1.2) in which explicit IVs are used.
- Keep using TLS 1.0, but introduce a single dummy first plaintext block in every TLS record. This dummy block can be for example an all-zero string.
- Keep using TLS 1.0, but send an empty message that has no data, but that would still result in adding only padding and MAC, i.e. the CBC-encrypted part of such a record will consist just of a MAC and padding.

The second and third options can be thought of as *hacks* to implementations of TLS 1.0 (and SSL). OpenSSL implemented the third option since version 0.9.6d, where a record with an empty plaintext fragment is prepended before sending the actual payload. The countermeasure was then switched off due to incompatibility with other implementations, mainly Microsoft Internet Explorer[21].

[21] http://www.openssl.org/~bodo/tls-cbc.txt

Some implementations, such as NSS opted for another hack to TLS 1.0, which relies on splitting non-empty application data records into two; the first record has *only* the first byte of plaintext, and the second has the rest.

Prioritising the RC4 algorithm over CBC-mode encryption was also proposed as another countermeasure. However, in 2013, the authors of [4] demonstrated that the TLS implementation of RC4 is vulnerable to other serious attacks, which we describe in the next section.

3.11 Other Attacks Against TLS

In this section, we give a short overview of recently published attacks against TLS.

3.11.1 Distinguishing Attack Against TLS with Short MACs

The authors of [74] described a distinguishing attack against the MEE-TLS-CBC construction. Their attack exploits the use of short MACs in TLS as standardised in RFC 6066 [2] and TLS's support for variable length padding. The outline of their attack is that if the size of the MAC is smaller than the size of the cipher block, and the plaintext message is small enough, then a distinguishing attack against TLS, with the MEE-TLS-CBC construction, can be mounted. The authors of [74] described how to *distinguish* whether an encrypted message contains for example `YES` or `NO` by modifying a few bits in the original ciphertext, C. The response (or lack of response) from the receiver of the TLS record helps the attacker identify whether the original message was `YES` or `NO`. The authors of [74] argue that the attack can be mounted in practice against TLS; they refer to the use of 80-bit truncated MACs in extensions to TLS 1.2, defined in RFC 6066 [2].

Countermeasures

The attack was considered to be more theoretical since no short MAC algorithms were supported in implementations of TLS. In addition, the work in [74] was not extended to a plaintext recovery attack. The recommended countermeasure would be not to use truncated MACs with the MEE-TLS-CBC construction.

3.11.2 The CRIME Attack

Recall that in Section 3.4.3, we described how TLS supports optional compression. The so-called CRIME attack was published in 2012 by the same authors of the BEAST attack [35]. The attack exploits the use of the DEFLATE compression method, implemented by the TLS Record Protocol and negotiated during the Handshake Protocol, in combination with a chosen plaintext capability to mount a plaintext recovery attack. The attack exploits side-channel information in the form of message size that is leaked when crafting web *requests* using a malicious web agent such as a JavaScript. All versions of TLS (and SSL) were vulnerable to the CRIME attack, regardless of their mode of operation.

Countermeasures

The workaround that was suggested and eventually deployed by most TLS implementations was to disable the use of TLS compression, i.e. implementations of TLS must make sure that the use of compression is not offered by the client in the Handshake Protocol's `ClientHello` message or is ignored by the server in case it was offered.

3.11.3 The BREACH Attack

The so-called BREACH attack [41] confirms the statement we made in the introduction chapter of the thesis that the interaction of secure network protocols with their upper and lower-layers plays a critical part in defining the system's overall security. Although implementations of TLS *disabled* the use of compression after the CRIME attack, as discussed earlier, the authors of the BREACH attack demonstrated how to exploit HTTP-level compression to mount a chosen plaintext attack that can recover TLS-protected plaintext. The BREACH attack relies on exploiting side channel information leaked in HTTP responses rather than requests. As expected, the attack applies to all versions of TLS with all modes of operation.

Countermeasures

A number of countermeasures are suggested in [41]. Examples of countermeasures listed in [41] include disabling HTTP compression, length hiding and limiting the number of cookie requests.

3.11.4 RC4 Attack

The authors of [4] present ciphertext-only plaintext recovery attacks against TLS when RC4 is selected for encryption. The authors of [4] identified new biases in the RC4 key stream output. In the multi-session setting, these biases can be used to recover plaintext with varying probability of success, depending on the position of the byte to recover and the amount of ciphertext captured. The authors of [4] also demonstrate how to exploit biases in consecutive pairs of bytes in the RC4 keystream that were first reported by Fluhrer and McGrew [37].

Countermeasures

A number of countermeasures have been proposed in [4]. It is worth noting that the attacks in [4] require large amounts of ciphertext and hence their practical relevance could be questioned. Despite this, countermeasures were implemented in practice to defeat the attacks. For example, Opera has implemented a cookie limiting countermeasure[22], while Microsoft has modified their code so that RC4 is no longer enabled by default for TLS in Windows 8.1 Preview[23].

3.12 Attacks Against DTLS

DTLS has not been put under as much scrutiny as TLS. This is largely attributable to the protocol's recent introduction and its limited, but growing, number of implementations, when compared to TLS. Most of the identified security issues with DTLS were associated with the protocol's implementation, with most of these issues resulting in a form of denial of service[24,25].

In addition to our work presented in Chapters 4 and 5, we have identified a number of security issues in one of Cisco Systems' DTLS implementations, which was based on OpenSSL. We successfully conducted denial of service and cipher suite downgrade attacks against the ASA line of products, in which DTLS is used to provide a remote access VPN. In fact, during this work we identified a critical OpenSSL software vulnerability that was independently discovered by another researcher and reported as CVE-2010-4180[26]. We communicated, privately, our findings to the vendor and worked with them to implement and test appropriate fixes.

We argue that our attacks against DTLS, which we present in later chapters, are by far the most involved and high-impact work carried out against the DTLS protocol, to date. In fact, performing a basic Internet web search for "DTLS vulnerability" reveals links mostly pointing to our work against DTLS.

[22] http://my.opera.com/securitygroup/blog/2013/03/20/on-the-precariousness-of-rc4
[23] http://technet.microsoft.com/en-us/library/dn303404.aspx
[24] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1386
[25] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4995
[26] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4108

3.13 Chapter Summary

In this chapter, we provided background information about the TLS and DTLS protocols. We discussed the structure of TLS and DTLS, their modes of operation, and explained the similarities and differences between TLS and DTLS. The TLS and DTLS protocols are flexible by design; new protocol extensions can be easily introduced, sometimes in an ad hoc fashion. Although this flexibility is an advantage, it could potentially introduce an amount of confusion to implementor who need to be familiar with the different versions of (D)TLS (and SSL 3.0), and their various extensions in order to implement and maintain the protocols properly, taking into consideration that all versions of (D)TLS (and SSL 3.0) are already in deployment.

We also covered a number of attacks against TLS that are relevant to the work we present in the next two chapters. The attacks we described in this chapter demonstrate the fact that *not* addressing what are considered at one point of time *theoretical* weaknesses could eventually lead to attacks that are practical and serious against a protocol, requiring ad hoc industry reaction (the BEAST attack being an example). As security researchers, we believe that "attacks only get better". Despite the number of high-profile attacks against the TLS MAC-then-Encode-then-Encrypt construction, and recently against RC4, we are yet to see a significant uptake of TLS 1.2 and authenticated encryption algorithms. Authenticated encryption algorithms deliver the two functions simultaneously: data encryption and authentication.

Chapter 4

Attacks Against DTLS

4.1 Introduction

Given the fact that the DTLS specification is based on that of TLS, implementations of DTLS should be immune to padding oracle attacks and their variants. In this chapter we show that this is *not* the case for either the OpenSSL or the GnuTLS implementations of DTLS 1.0. More specifically, this was not the case in OpenSSL versions 1.0.0e or 0.9.8r and earlier, and GnutLTS version 3.0.10 and earlier, the latest versions of these libraries at the time we carried out our research. DTLS 1.2 [83] was an RFC draft at the time when we carried out our work and was not implemented by OpenSSL or GnuTLS.

In this chapter, we first focus on OpenSSL, showing that there is a small timing difference in OpenSSL's processing of DTLS packets having valid and invalid padding fields: just like old versions of OpenSSL's implementation of TLS, if the padding is invalid, then the MAC is not checked, while if the padding is valid, the MAC check is done. This results in a timing difference for processing of packets with valid and invalid padding that is on the order of a few tens of microseconds (μs) on a modern processor.

However, one major difference between TLS and DTLS, and which we highlighted in Chapter 3, is that DTLS provides no error messages when decryption encounters an error. The detection of these error messages is *essential* to the attacks of Canvel *et al.* [25] on TLS. Thus it would appear that this timing difference *cannot* be used to build a padding oracle. This may explain why the OpenSSL code for DTLS had not been patched to remove the known timing difference.

By bringing new techniques into play, we show that the lack of DTLS error messages is not a serious impediment to the attack – we are able to exploit the DTLS extension for Heartbeat messages [93] to ensure that the timing difference shows up in the timing of Heartbeat response messages rather than error messages. In fact, any upper layer protocol which has messages that also provoke a response message with a predictable delay can be used in place of Heartbeat messages in our attack. We demonstrate this in our attack against GnuTLS where we use other types of message than Heartbeats. We also introduce new techniques which *amplify* the identified timing difference. In TLS, this is easily done by using long messages, since TLS supports messages up to roughly 2^{14} bytes in size. This approach was used by Canvel *et al.* [25]. But this is not possible in DTLS, since the maximum message size is limited by the path maximum transmission unit (PMTU). To overcome this, we build trains of DTLS packets which all either have valid or invalid padding and hence which all contribute to an accumulated timing difference in the same way. These trains need to be carefully injected into the network – fast enough so as to ensure each packet arrives before the processing of the previous one has completed, but not so fast that DTLS's buffer for incoming packets gets swamped. Thus the success of the attack depends on delicate, μs-level timing of network events.

Another major difference we highlighted in Chapter 3 in between TLS and DTLS is that, in TLS, any error arising during cryptographic processing is treated as fatal, meaning that the TLS connection is discarded in the event of any error. TLS can afford to do this because it is built on top of a reliable transport protocol, TCP. DTLS, on the other hand, cannot afford to do so, since its underlying transport protocol is UDP. This means that DTLS does *not* discard connections in the event of errors, but merely discards error-generating packets. The reader may refer to Section 3.5 for a detailed description of the differences between TLS and DTLS. So, in contrast to previous attacks on TLS, our attack on OpenSSL's DTLS implementation can efficiently recover as much plaintext as the adversary desires, without having to wait for the re-establishment of DTLS connections. This also underlies the above mentioned amplification technique. Our attack becomes even more efficient in the situation where DTLS's anti-replay feature is disabled, which is an option within the DTLS specification.

We then switch our focus to the GnuTLS implementation, and show that, even though it properly implements the countermeasures in TLS 1.1, it is still vulnerable to a partial plaintext recovery attack in its default configuration. We show that a small timing channel is introduced into the decryption process because a plaintext-dependent sanity check is carried out at an early stage during decryption, followed later by assigning a zero value to the plaintext message length in the case when this sanity check fails. This introduces a detectable timing difference that, when combined with our new techniques, allows 4 or 5 bits of plaintext to be recovered per ciphertext block. In principal, the attack could also be applied to the GnuTLS implementation of

TLS (but then the timing difference would be hard to amplify).

Despite the availability of easy fixes, we argue that the attacks are still interesting and provide valuable lessons for protocol designers and implementors:

- To our knowledge, our attacks are the first of their kind against any implementations of DTLS. Our OpenSSL attack is also the first plaintext-recovering attack against a protocol implemented by OpenSSL since the work of Canvel *et al.* [25].
- Our attacks exploit the fact that DTLS has to be error-tolerant, but we had to find novel means to circumvent the resulting lack of error messages.
- The DTLS specification (for versions 1.0 and 1.2) is rather brief and refers to the TLS specification for many details, particularly those relating to how packets are encrypted and decrypted. This then requires an implementor to cross-refer to other standards during implementation, which may lead to software that does not implement the known countermeasures.
- Our attack on the GnuTLS implementation of DTLS and TLS shows that, even if all the known countermeasures are carefully implemented, DTLS and TLS implementations may still be vulnerable to attack via subtle timing side channels.
- Our attack on GnuTLS also points the way forward to the more general attacks in the next chapter.

We expand on these themes later in the chapter. Section 4.2 presents our basic attack against the OpenSSL implementation of DTLS. Section 4.3 discusses a number of implementation issues for this attack and discusses refinements of it. Section 4.4 presents our experimental results demonstrating efficient and reliable recovery of full DTLS plaintexts in the OpenSSL case. Section 4.5 briefly discusses how similar attacks can recover partial plaintexts in the GnuTLS case. Section 4.7 discusses the wider implications of our work for secure network protocol design.

4.2 Building a Padding Oracle for OpenSSL

4.2.1 Using the Heartbeat Extension

Although we exploit Heartbeat messages in our attack against OpenSSL, other type of messages could also be used. The only constraint is that they should always predictably generate responses that can be detected by the adversary. We demonstrate this in our attack against GnuTLS.

4.2.2 Assumptions on the Adversary

The objective of the attack is to recover DTLS-protected plaintext. We assume that the adversary:

- Has access to the ciphertext. This can be achieved by the adversary gaining access to a network device like a switch or a router and making copies of the ciphertext exchanged by the endpoints of a DTLS connection.
- Can send arbitrary DTLS messages to the original recipient. This can be achieved by injecting packets into the network while spoofing the IP and UDP headers.
- Is aware of the encryption algorithm's block size, b. The adversary can infer this by either monitoring the connection's handshake messages, or the size of the encrypted messages over time.
- Can detect and record a number of Heartbeat request packets.

The above assumptions apply when anti-replay is deactivated. We note that anti-replay is enabled by default for both the OpenSSL and GnuTLS implementations of DTLS, and we had to modify the server source code to disable it in our experiments. When anti-replay is activated, then we also need to assume that the adversary can stop messages of his choice from reaching their final destination. For example, the adversary may achieve this by exploiting his control over a router or a firewall in the data path. In presenting our attack below, we assume that anti-replay is disabled, i.e. we assume that the targeted system does *not* perform sequence number checking for incoming DTLS messages. We explain how to modify the attack to handle the case where anti-replay is enabled in Section 4.3.6.

4.2.3 Building a Padding Oracle for the OpenSSL Implementation of DTLS

In this section, we explain how to construct a padding oracle for the OpenSSL implementation of DTLS. This oracle can then be used in the standard way to decrypt arbitrary ciphertext blocks and hence arbitrary amounts of plaintext data, as described in Section 3.9 of Chapter 3. The key observation we use is that, in the OpenSSL implementation of DTLS (the latest the versions available at the time we carried out our research), if the padding underlying a ciphertext is valid, then the MAC on the message is checked, whereas if the padding is invalid, then the MAC is not checked and the ciphertext is rejected immediately. This contravenes the requirement for equal processing times in TLS 1.1 that is inherited by reference in the DTLS specification.

As a consequence of this deviation, we would expect the processing time for a DTLS packet with invalid padding to be slightly less than that of a DTLS packet with valid padding. The actual time difference depends on a number of factors including the algorithms used, the clock-speed of the target system, the size of the DTLS packet, other processes running on the target system, and the network conditions. For example, we measured the MAC verification time on our testing machine running OpenSSL with HMAC-SHA-1 and found it to be in in the order of tens of μs – see Figure 4.1.

So far, this is identical to the timing side channel exploited in [25]. However, DTLS does *not* have any error messages, so we cannot use existing methods to observe the difference in processing times. This may explain why the implementors of DTLS in OpenSSL chose not to implement the required countermeasures. Instead, we introduce an alternative means of observing the difference, by exploiting Heartbeat messages. The basic idea is quite simple. Suppose we send to the target system a packet train consisting of a DTLS packet P_C carrying the ciphertext C (whose padding validity we wish to test) immediately followed by a Heartbeat request message. Then this train will result in a detectable Heartbeat response message being sent back on the network, and, assuming orderly processing on the target system, the total amount of time needed to process P_C and to produce the Heartbeat response message will reflect whether or not MAC verification was carried out when processing C. From an adversary's perspective, only send and receive times of packets can be captured, so the adversary will measure the time difference between sending the initial packet train and receiving the Heartbeat response packet, which we refer to as the *round trip time (RTT)*. If this time difference is larger than some threshold T, the adversary will assume the padding was valid (and so the MAC verification was carried out), while if it is lower than this threshold, the adversary will assume the padding was invalid. The threshold can be set by doing some initial system profiling to measure the typical timing difference between packets carrying ciphertexts having valid and invalid padding. Notice also that DTLS Heartbeat packets are not essential to building the oracle: any upper layer protocol having suitably predictable and detectable response messages can be used.

In reality, the timing of packets is influenced by many factors beyond just DTLS's cryptographic processing. Moreover, as we noted above, the timing difference will be rather small for normal-sized packets. So the DTLS padding oracle as presented would be much too error-prone. To enhance the accuracy of the oracle, the adversary can:

- Choose a specific, favourable DTLS packet payload length, l.
- Send n copies of packet P_C in a train followed by a Heartbeat request instead of just one copy of P_C. Here, the idea is that each copy of P_C will be processed

Algorithm 3: Padding Oracle for OpenSSL implementation of DTLS

```
Data: $C$
Result: VALID or INVALID

for $q = 1$ to $m$ do
    $RTT_q$ = Timer($C$);
$RTT$=Mean($RTT_1, RTT_2, ..., RTT_m$);
if $RTT \geq T$ then
    return VALID;
else
    return INVALID;

Timer($C$)
Set $T_s$ = current time;
Send $n$ copies of $P_C$, a DTLS packet containing $C$, to the targeted system;
Send a Heartbeat request packet to the targeted system;
Set $T_e$ = time when Heartbeat response packet is seen;
return $(T_e - T_s)$
```

in the same way, so the larger the accumulated time difference will become and the easier it will become to distinguish between valid and invalid padding. This exploits the fact that DTLS does not tear-down DTLS connections in the event of errors (recall that when the padding oracle is used in a plaintext recovery attack, all the ciphertexts sent in the attack will be invalid in some way – they will either have invalid padding or invalid MACs). It also assumes that all the packets in the train can be made to arrive at the target system in such a way that no adverse delays are introduced during the processing of these packets.

- Send m packet trains (each containing n copies of P_C), and use a suitable statistical model to analyse the observed RTTs.

Algorithm 3 describes our basic DTLS padding oracle for a ciphertext C. In the algorithm, RTT_q denotes the response time in the q-th trial, T denotes the threshold for deciding on whether C has valid or invalid padding, and simple averaging is used to process the gathered RTTs. Other statistical measures could be used in place of averaging here, an idea that we discuss in more detail in the next section. There, we also explore the many practical issues that arise in building this padding oracle, addressing issues such as packet timing, system profiling, parameter selection to tune the attack, and dealing with anti-replay.

4.3 Practical Considerations

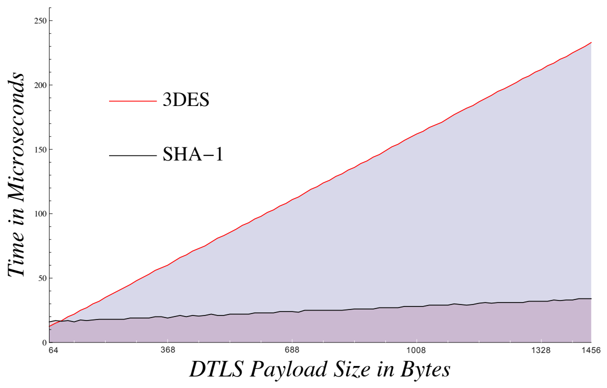
In this section, we discuss a number of practical issues that arise in implementing our attack. All of our remarks are specific to the OpenSSL implementation of DTLS.

4.3.1 Timing and OpenSSL Cryptographic Operations

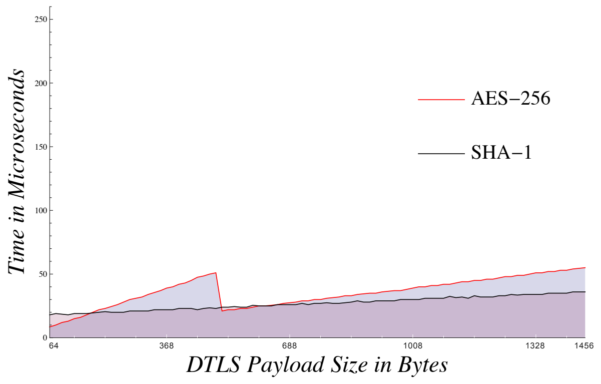
Our attack relies on detecting the time difference introduced by MAC verification that is performed for packets having valid padding but not for packets having invalid padding. Failure to detect this time difference would result in the padding oracle providing an incorrect answer. Figure 4.1 shows, for a variety of DTLS payload sizes, the time taken by OpenSSL in our set-up to perform decryption with 3DES or AES-256 alongside the time taken for MAC verification using HMAC-SHA-1. The hardware specifications of our set-up are listed in Section 4.4. We note the following features evident from this figure:

- In general, decryption is slower than MAC verification, especially in the case of 3DES.
- The MAC processing time for a single packet is on the order of a few tens of μs, which is well below that reported in [25] and below the level of jitter expected in a typical network.
- 3DES is much slower than AES-256: for a packet size of 1456 bytes, the factor is about 4. For reasons that will be explained below, using a slower decryption algorithm increases the effectiveness of the attack. Hence the attack parameters (l, m, n) may need to be tuned depending on which block cipher is in use.
- With AES-256, the processing time rapidly drops from about 50 μs to about 20 μs when the DTLS payload size reaches 512 bytes. We do not know the exact reason for this behaviour, but the adversary also needs to be aware of it when selecting attack parameters. One possible explanation is that a switch to a more efficient AES implementation is made once the payload is sufficiently large.

Although we have targeted HMAC-SHA-1 in our attack, the fundamentals of the attack still apply when other MAC algorithms are in use. At the time of carrying out this work, OpenSSL only supports HMAC-MD5 and HMAC-SHA-1. More detail about how packets are processed and the source of the timing difference is provided in the next section.



(a) 3DES and HMAC-SHA-1



(b) AES-256 and HMAC-SHA-1

Figure 4.1: Timing of cryptographic operations for DTLS payloads of sizes between 64 and 1456 bytes. With AES-256, the processing time rapidly drops from about 50 μs to about 20 μs when the DTLS payload size reaches 512 bytes. One possible explanation for this behaviour is that a switch to a more efficient AES implementation is made once the payload is sufficiently large.

4.3.2 Timing and Packet Processing

Let us look in detail at how a receiver processes a packet, with a view to building a simple model of how RTTs are affected by the attack parameters. To this end, Figure 4.2 shows a simplified time-line of how packet i, having valid padding, is processed by the receiver.

In the time-line we have:

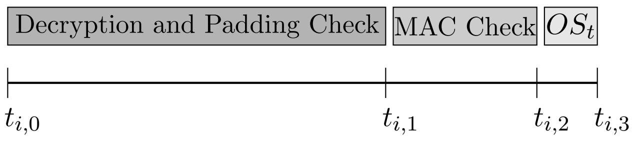


Figure 4.2: Packet processing time-line – valid padding: $t_{i,0}$ is the time at which packet i arrives in the OpenSSL buffer, $t_{i,1}$ is the time at which the decryption and padding check are completed for packet i, $t_{i,2}$ is the time at which the MAC check is completed for packet i, $t_{i,3}$ is the time at which OpenSSL is ready to process the next DTLS packet, packet $i+1$, and OS_t is any additional time spent by the operating system in relation to the processing of the packet (we assume this to be a constant, independent of i).

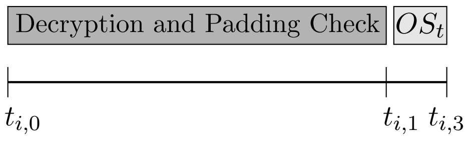


Figure 4.3: Packet processing time-line – invalid padding. In the case of a packet with invalid padding, the MAC verification is not performed and hence we have $t_{i,2} = t_{i,1}$.

- $t_{i,0}$: The time at which packet i arrives in the OpenSSL buffer. The buffer holds DTLS packets waiting to be processed.
- $t_{i,1}$: The time at which the decryption and padding check are completed for packet i.
- $t_{i,2}$: The time at which the MAC check is completed for packet i.
- $t_{i,3}$: The time at which OpenSSL is ready to process the next DTLS packet, packet $i+1$.
- OS_t: Any additional time spent by the operating system in relation to the processing of the packet. We assume this to be a constant, independent of i.

In the case of a packet with invalid padding, the MAC verification is not performed and hence we have $t_{i,2} = t_{i,1}$. Figure 4.3 is the analogue of Figure 4.2 for the case of invalid padding, and illustrates that, for a fixed DTLS packet length, the time taken to process a packet with invalid padding is less than that taken to process a packet with valid padding.

In Section 4.2, we defined *RTT* to be the time taken between sending the first packet in a train to receiving a Heartbeat response packet. Next, we analyse the

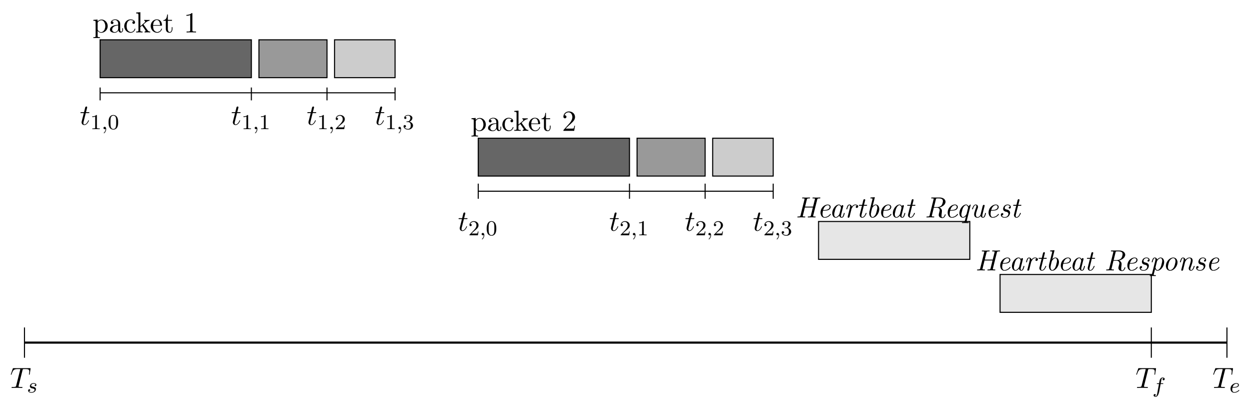


Figure 4.4: Time-line for a train with $n = 2$ (not to scale).

different contributions to RTT. As an example, the time-line in Figure 4.4 shows a train made of two identical data packets (so $n = 2$), both having valid padding, followed by a Heartbeat request packet, which then provokes a Heartbeat response packet. In Figure 4.4 we have:

- T_s: The time at which the adversary sends the first DTLS packet, packet 1.
- T_f: The time at which the Heartbeat response packet is sent by the receiver
- T_e: The time at which the Heartbeat response packet is received by the adversary.
- $t_{1,0} - T_s$: The time it takes for packet 1 to reach the receiver.
- $T_e - T_f$: The time it takes for the Heartbeat response packet to reach the adversary after being sent by the targeted system.
- $T_e - T_s$: The RTT for the packet train.

Figure 4.4 shows the second data packet, packet 2, arriving *after* the completion of processing of packet 1, i.e. so that $t_{2,0} > t_{1,3}$. The same applies to the Heartbeat request packet arriving after the completion of processing of packet 2. In this situation, the receiver enters a wait state until the next packet arrives and the arrival time of a packet and its processing start time are the same. In general, this situation results in some or all of the timing difference arising because of the MAC verification being "absorbed" into the wait state of the receiver, and hence is sub-optimal in terms of detecting the time difference.

In the opposite situation, where packet 2 arrives before processing of packet 1 is complete, packets are buffered. Then packet 2 is immediately available for processing

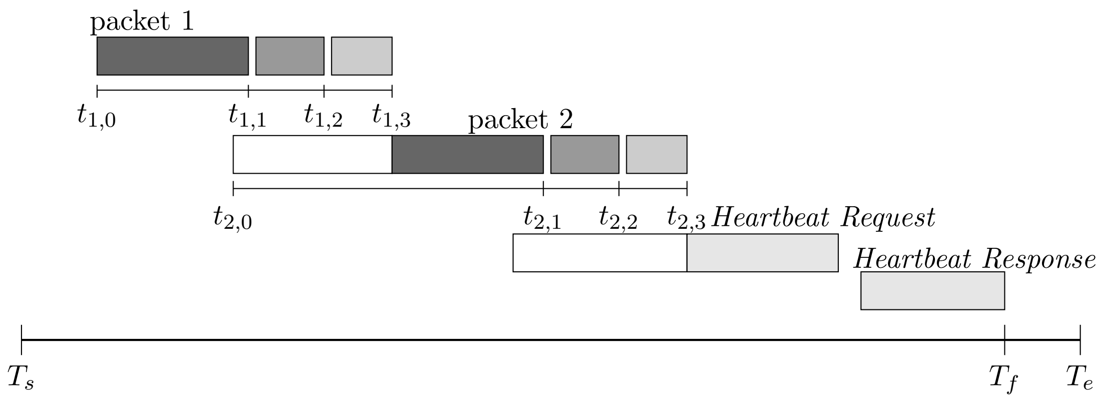


Figure 4.5: Time-line for packet train with valid padding and packet buffering.

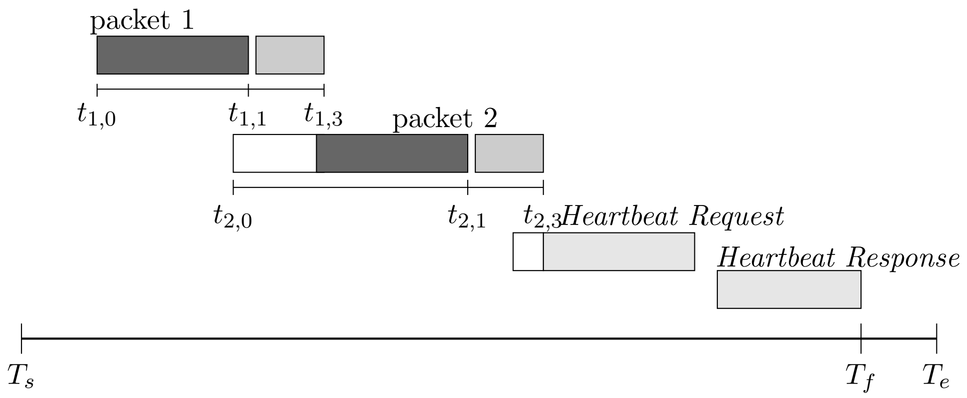


Figure 4.6: Time-line for packet train with invalid padding and packet buffering.

at the receiver as soon as processing of packet 1 is complete, and none of the MAC verification time is absorbed. The buffer is managed by OpenSSL and its maximum size is 100 DTLS packets. Figures 4.5 and 4.6 illustrate this situation for packet trains having valid and invalid padding, respectively, with the white boxes representing the amount of time spent by packets in the buffer. It is evident from these figures how the time arising from MAC verification (in the case of valid padding) accumulates packet-by-packet to create an amplified time difference in the RTT for the train.

The upshot of this analysis is that, from the adversary's perspective, it is desirable to select the attack parameters so that the receiver's buffer always contains some (but not too many) packets. In this way, the receiver is never waiting for a packet to arrive and the MAC processing time accumulates across the whole packet train.

We have experimentally verified the essential basic correctness of this model for packet processing in the following way. Let RTT_1 denote the RTT for a train that uses packets having valid padding (and for which the MAC is verified), and let RTT_2 denote

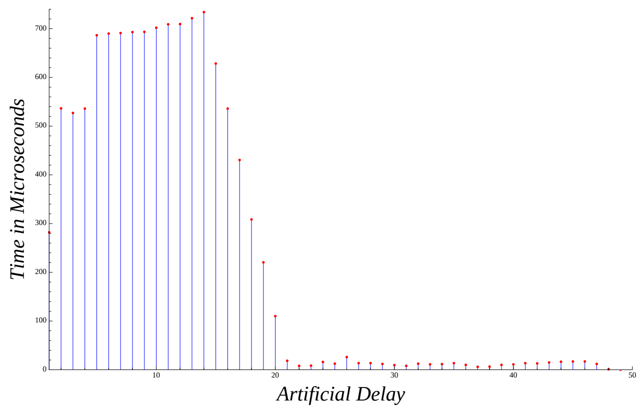


Figure 4.7: Value of δ, the difference in RTTs for valid and invalid padding, against artificial delay. The figure confirms that injecting artificial delays in between packets from the train as they leave the adversary's machine, results in the value of δ steadily decreasing and eventually reaching zero as the size of the artificial delay increases.

the RTT for a train that uses packets having invalid padding. Let δ denote the time difference between the two RTTs, so that:

$$\delta = RTT_1 - RTT_2$$

Then, if we artificially inject delays in between packets from the train as they leave the adversary's machine, we would expect to see the value of δ steadily decrease and eventually reach zero as the size of the artificial delay increases. Figure 4.7 shows the results of such an experiment which confirms this behaviour. Somewhat surprisingly. This is because, as the artificial delay increases, more and more of the extra time introduced into RTT, by MAC processing is absorbed in waiting for the next DTLS packet in the train to arrive. Figure 4.7 also shows that adding small artificial delays can actually increase the time difference δ, making this difference in RTTs *easier* for the adversary to detect. We do not have an explanation for this effect.

4.3.3 System Profiling

System profiling refers to the process by which the adversary collects information about the targeted system prior to carrying out an attack. This provides the adversary with the expected values for the RTTs (for valid and invalid padding) under some conditions such as system load, the DTLS payload length, l, and the number of packets in the train, n. This profiling in turn allows the threshold value T for the attack to be set.

Given a captured ciphertext, it is easy to construct ciphertexts having any desired length l and having either invalid or valid padding, simply by manipulating the last 2 blocks of the captured ciphertext and prepending random blocks (or truncating it if a shorter ciphertext is needed). Given such pairs of ciphertexts and a Heartbeat request message, the adversary can then construct packet trains containing the required number of packets n. These trains can then be repeatedly sent to the target system and the RTTs measured, to obtain two empirical probability density functions (PDFs), one for trains with validly padded packets and the other for trains with invalid padding. From these PDFs, the threshold T can be set by, for example, calculating the mean of each distribution and setting T to be the mid-point between the means. In practice, we tend to obtain small numbers of extreme outliers in such profiling experiments, and removing these before calculating the means by using a simple cut-off generally improves the performance of the attack. More sophisticated statistical methods can of course be employed, but we have found profiling followed by thresholding to be already adequate for our attacks to be successful.

4.3.4 An Attack without System Profiling

System profiling is not even strictly necessary – for a given byte position i in the target block, an adversary can simply measure the RTTs for a packet train (consisting of n DTLS packets with the target ciphertext block being located at the end of each packet, followed by a Heartbeat request packet), for each of the 256 possible byte values in position i in the ciphertext block preceding the target ciphertext block. Then the adversary can select as the correct byte value (i.e. the one giving valid padding) the one that maximises the RTT across the 256 measured RTT values. Accuracy can be further improved by repeating the trial for each byte m times, removing outliers, and using the maximum of the average RTTs. In fact, we have observed in our experiments that repeating the trial for each byte value m times, removing outliers, and then selecting the byte value that maximises the *minimum* of the m measured RTTs for each byte value gives substantially higher success probabilities for the attack. We will illustrate this in Section 4.4 where we discuss our experimental results in more detail. This, then, is the preferred version of our attack. Note that, strictly speaking, this version of the attack does not build a padding oracle, but rather considers all possible 256 byte values simultaneously.

Even more sophisticated statistical techniques, such as sequential estimation (as in [25]) or likelihood estimation, can be used in place of averaging or selecting the minimum when processing the results of the m trials per byte. However, these more advanced approaches were not needed in order to successfully launch our attack. They could be useful in further reducing the amount of data sent or the number of Heartbeat request messages consumed in an attack.

Finally, we note two further advantages of using an attack without profiling. Firstly, the process of profiling itself will require Heartbeat request messages to be gathered. Secondly, the attack environment may change over time during the attack itself, as varying network or server loads are experienced, for example. The attack without profiling described here automatically adjusts for such changes, at least if they do not occur within the time taken to recover a single byte of plaintext.

In Section 4.4, we describe the results of our implementation of the attack, where profiling was *not* conducted.

4.3.5 Measuring Success Under Budgetary Constraints

The attack is such that a byte is successfully decrypted only if all the preceding bytes in the same block are successfully decrypted. Hence, under a reasonable independence assumption, if the probability of successfully decrypting a byte is p, then the probability of successfully decrypting a block of size b will be $p_b = p^b$. For AES, $b = 16$, so for successful decryption of a whole block with a reasonable probability, we need p to be rather close to 1. For example, with $p = 0.99$ and $b = 16$ we have $p_b = 0.85$. The adversary can tune the attack parameters (l, m, n) so as to increase the success probability p of the attack and can try to find the optimal combination that results in the highest success probability. However, in practice, an adversary will have a limit on, for example, the maximum number of bytes that he wishes to send in order to recover a byte. As discussed below, when anti-replay is enabled, Heartbeat request packets (or their equivalents) will become a precious resource. Since each train consumes one such packet in this situation, it may be desirable to increase l, the packet size and n, the number of packets per train, so as to maximise the amplification effect, whilst minimising m, the number of trains sent per byte. However, as our later experimental results will show, simply increasing l and n does not always help, especially in the case of AES-256.

4.3.6 Attacks with Anti-Replay Enabled

Attacking DTLS becomes slightly more complex when anti-replay is enabled. Since the OpenSSL implementation of DTLS first checks the sequence number against the anti-replay window before doing any cryptographic processing, the adversary has to take care that all packets sent in trains do not have sequence numbers that are marked as having previously arrived. Fortunately, the anti-replay window is only updated if the MAC on a packet is successfully verified, and all the packets used in the attack will fail the MAC verification (with the exception of the Heartbeat packets), so the window is not updated as a consequence of these attack packets.

With anti-replay enabled, each Heartbeat request packet can be used only once, since its sequence number will be marked in the window as having been seen once the packet arrives. Moreover, the adversary has to ensure that the sequence number for each Heartbeat request packet used does fall within (or to the right of) the current anti-replay window, otherwise the Heartbeat request will be discarded and no response generated.

Thus Heartbeat request packets become a precious resource in the situation where anti-replay is enabled: the attack can only proceed as quickly as they become available. Hence decryption in this setting may be rather slow and "opportunistic" – every time a packet is seen on the wire by the adversary, a new packet train can be launched and a byte value tested.

Given these issues, it is apparent that the adversary should try to use as few Heartbeat request packets as possible, which means minimising m for a given target success probability p. A further enhancement arises by building packet trains that test multiple byte values *simultaneously*. For example, the adversary could build two sets of m trains, each train containing $128n$ packets, with half of the possible byte values being tested in each train n times each. This would represent the first step in a binary search for the correct byte value, requiring only 8 steps and therefore $16m$ Heartbeat request packets to extract a byte. The number of Heartbeat requests consumed could be halved again with initial system profiling. In contrast, our basic attack would consume $256m$ Heartbeat request packets for the same result. We have not tested this version of the attack, but our experience indicates that it would work well whenever using long packet trains does not degrade performance.

Finally, we recall that packets from *any* suitable application layer protocol could be used in place of Heartbeat request packets, so long as the corresponding application always sends a detectable response packet with a predictable response time. So the success of our attack does not depend completely on the availability of Heartbeat request packets in the case where anti-replay is enabled.

4.4 Implementation and Results for OpenSSL

4.4.1 Implementation

In our laboratory set-up, we have a client, the adversary and the targeted system all connected to a 100Mbps Ethernet switch on the same VLAN. The targeted system was a machine running a single core processor operating at a speed of 1.87 GHz and having

2 GByte of RAM.

We ran version 1.0.0a of OpenSSL on the client and the server. We used the built-in OpenSSL utilities for the client[1] and the server[2], `s_client` and `s_server` respectively. `s_client` implements a generic client which connects to a remote host using DTLS, while `s_server` implements a generic server which listens for connections on a given UDP port using DTLS. We implemented the Heartbeat extension feature by installing the appropriate OpenSSL patch[3]. For experimental convenience, we deactivated anti-replay, by directly modifying the OpenSSL code, i.e modifying the default behaviour of OpenSSL.

4.4.2 Results

The results shown in this section reflect our specific set-up. Of course, the values would change as the set-up changes – for example, the timings are heavily dependent on the clock-speed of the processor used on the target system. However, the fundamentals of the attack would remain the same.

Experimentally observed s:

The figures we discuss hereafter show PDFs[4] observed in our experiments for different attack parameters and encryption algorithms. In all the figures, the x-axis represents RTTs while the y-axis represents the probability of observing these RTTs. In all figures, outliers have been removed. Each figure shows two PDFs, PDF_1 (in red) and PDF_2 (in blue), that correspond to having valid and invalid padding in the packets in the trains, respectively. We recall that l denotes the DTLS payload size, m denotes the number of trials per byte, and n denotes the number of DTLS packets per trial. Figures 4.8 and 4.9 show PDFs for n equal to 10 and varying the value of l, for 3DES and AES-256 respectively. We note the following:

- It is generally easier to distinguish between the two PDFs in the case of 3DES when compared to AES-256, shown in Figures 4.8 and 4.9 respectively.
- Generally, there is an increasing overlap between the two PDFs as the value of l, the DTLS payload size, increases. This is more evident in the case of AES-256.

[1] http://www.openssl.org/docs/apps/s_client.html

[2] http://www.openssl.org/docs/apps/s_server.html

[3] http://sctp.fh-muenster.de/dtls-patches.html

[4] The PDF figures shown in this chapter and the next chapter are mostly smooth histograms, i.e. PDFs that are based on a smooth kernel density estimate.

- In the case of AES-256, increasing l makes the PDFs much harder to distinguish. The reason for this is that the adversary spends more time preparing and sending packets as the packet size increases, while the targeted system may already have finished AES decryption and MAC verification and be waiting for the next packet. Thus long packets tend to arrive “late” at the targeted system.

Figures 4.10 and 4.11 show the PDFs for $l = 1024$ and varying the value of n, for 3DES and AES respectively. We note the following:

- In the case of 3DES, increasing the value of n helps in making the two PDFs more distinguishable. This is the case with AES-256 when small DTLS payloads are used.

- With AES, increasing the value of n when using large DTLS payloads makes the PDFs harder to distinguish. Figures 4.12 and 4.11 show this effect when AES-256 is used for $l = 256$ and $l = 1024$ respectively.

- By appropriately choosing the attack parameters, it is possible to obtain PDFs that are very easy to distinguish. For example, the last graph in Figure 4.12 shows the PDFs for AES-256 when $l = 256$ and $n = 160$, where the peaks are separated by more than 500μs while the distributions are entirely contained within 50 μs of the peaks.

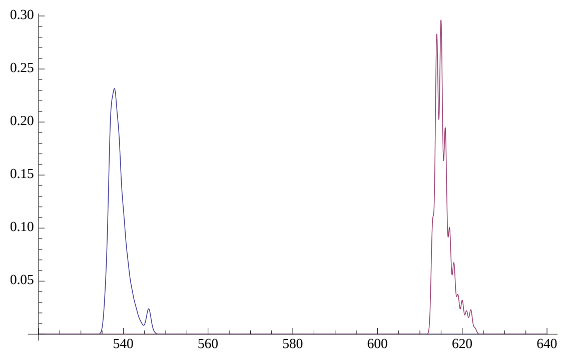
Success Probability:

Table 4.1 shows the success probability, p, of decrypting a byte under different attack parameters (l, m, n) when AES-256 is used. We recall that the success probability for a block is then given by p^b where b is the block length in bytes.

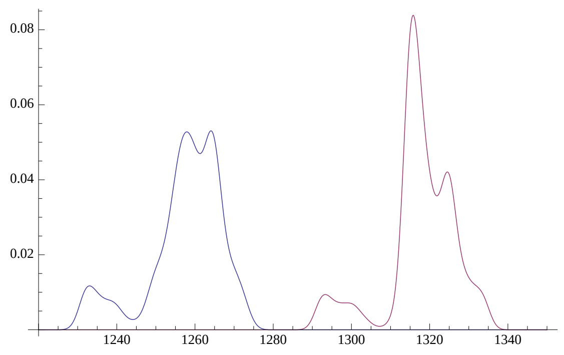
These tables were obtained using the preferred version of our attack described in Section 4.3.4, where no system profiling is used, outliers are removed, and, for each byte, we use the minimum RTT value from the m values available, and then select the correct byte as being the one that gives the maximum amongst these values. Each entry in the tables is calculated using 100 runs of the attack.

We can clearly see that the probability of success increases as the number of trials, m, increases. Success probabilities p equal to 0.99 or above are easily achieved for moderate values of l, m and n, making our preferred attack both efficient and highly reliable for these parameter choices.

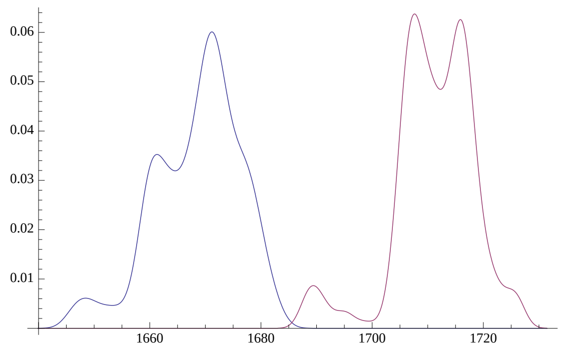
Table 4.2 shows analogous success probabilities for 3DES. Note however that in these tables, we report figures for substantially larger values of l than we did for AES-256. This is indicative of the fact that our attacks are still quite successful for 3DES



(a) $l = 256$



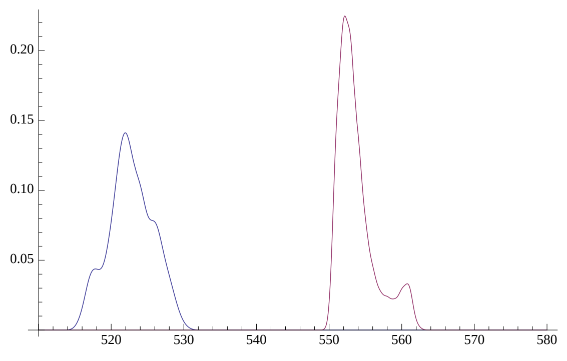
(b) $l = 1024$



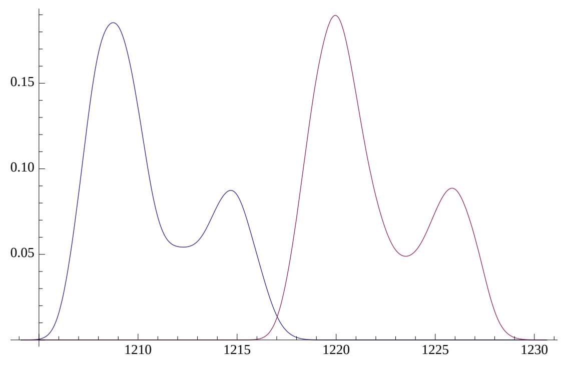
(c) $l = 1456$

Figure 4.8: 3DES – PDFs for $n = 10$ and varying l (packet size), with outliers removed. In this figure we show two PDFs, PDF$_1$ (in red) and PDF$_2$ (in blue), that correspond to having valid and invalid padding in the packets in the trains, respectively. In the case of 3DES, we can easily distinguish between the two PDFs.

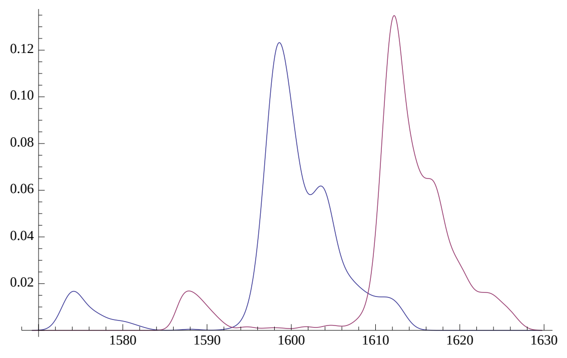
even with long payloads, giving an additional amplification opportunity. As further confirmation of the practicality of our attacks, Table 4.3 provides success probabilities for AES-256 for $l = 192$ and various values of m and n, with the probabilities being based on 1000 runs of the attack. For example, already for $m = 10$ and $n = 2$, the success probability is 0.996, meaning that an entire block of plaintext can be recovered



(a) $l = 256$



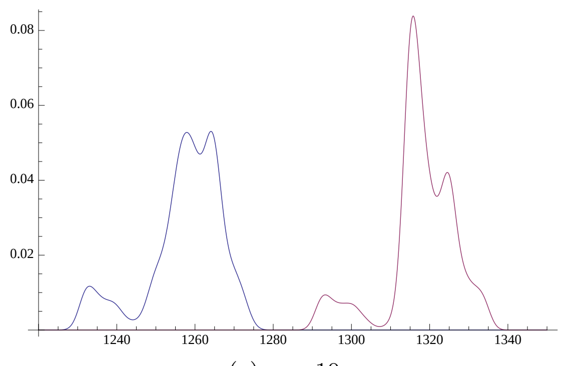
(b) $l = 1024$



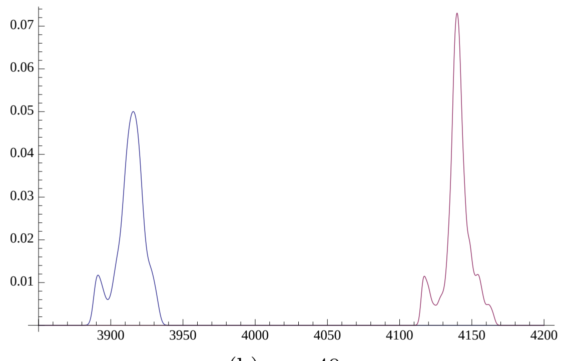
(c) $l = 1456$

Figure 4.9: AES-256 – PDFs for $n = 10$ and varying l (packet size), with outliers removed. In this figure we show two PDFs, PDF_1 (in red) and PDF_2 (in blue), that correspond to having valid and invalid padding in the packets in the trains, respectively. In the case of AES-256, It is harder, but possible, to distinguish between the two PDFs, when compared to 3DES shown in Figure 4.8.

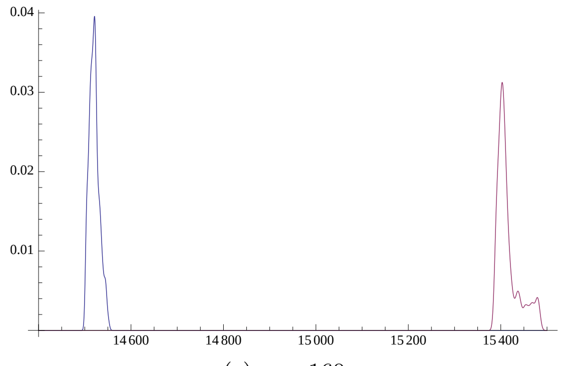
correctly with probability 0.94, at a cost of (roughly) 7000 bytes of network traffic per byte.



(a) $n = 10$



(b) $n = 40$



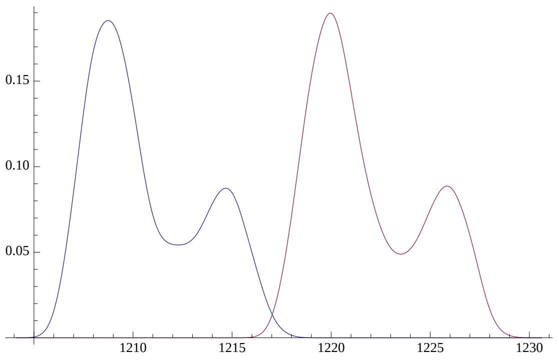
(c) $n = 160$

Figure 4.10: 3DES – PDFs for $l = 1024$ (packet size) and varying n (train size).

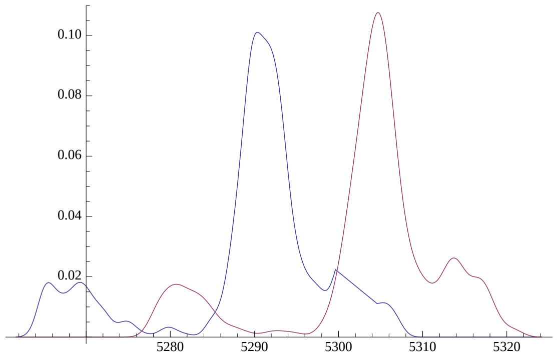
4.5 Attacking the GnuTLS Implementation of DTLS

We have examined the GnuTLS implementation of DTLS, with the intention of finding similar attacks. However, the code for decryption[5] is such that there is no timing difference for processing of packets with valid and invalid padding: the MAC verification is carried out in either case, and only then is the packet dropped. However, the code does

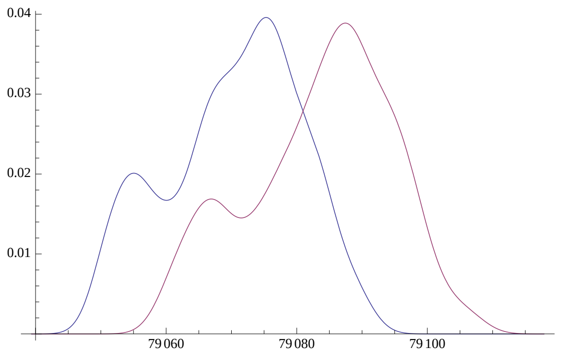
[5]See `http://git.savannah.gnu.org/gitweb/?p=gnutls.git;a=blob;f=lib/gnutls_cipher.c`



(a) $n = 10$



(b) $n = 40$



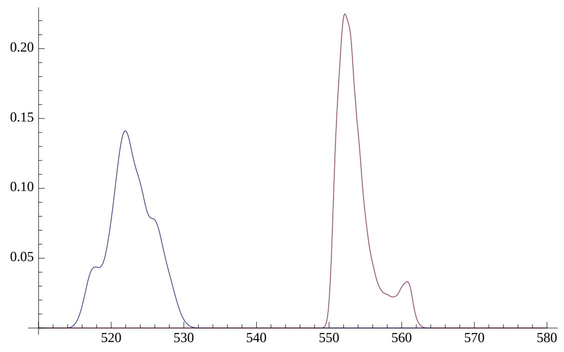
(c) $n = 160$

Figure 4.11: AES-256 – PDFs for $l = 1024$ (packet size) and varying n (train size).

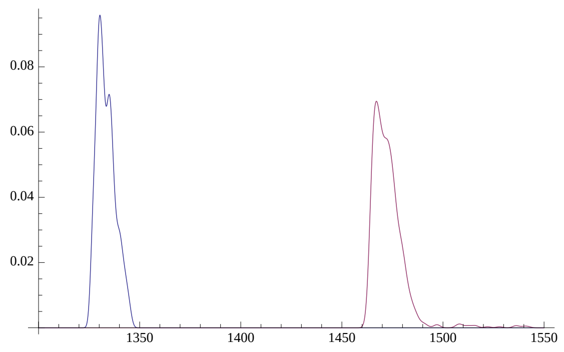
include the lines shown in Figure 4.13 that are executed after CBC-mode decryption.

From this code, specifically line 552, it can be seen that if a certain test involving the padding length in `pad` and the ciphertext size fails, then `pad_failed` is set to `GNUTLS_E_DECRYPTION_FAILED`, which is a negative integer, and the variable length (which would otherwise be negative) is set to 0.

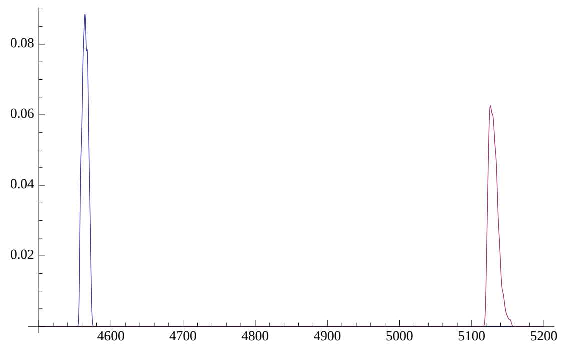
The rest of the packet processing then proceeds as normal (but with a padding check not being performed and `length` being set to `0`. The time taken to process a packet



(a) $n = 10$



(b) $n = 40$



(c) $n = 160$

Figure 4.12: AES-256 – PDFs for $l = 256$ (packet size) and varying n (train size).

when the test fails and `length` is set to `0` should be less than that taken to process a packet when the previous value for `length` is maintained. This is because, when `length` gets set to `0`, no padding check is performed, and the MAC verification performed in lines 582 to 593 is done on a smaller amount of data (effectively, just the 13 bytes of sequence number and header data). Each packet that fails the padding length test or the padding check results in GnuTLS printing a "`Discarded message due to invalid decryption`" error message to the screen. Unless the debugging level is changed, no

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.00	0.15	0.41	0.32	0.00	0.01
2	0.02	0.24	0.32	0.40	0.00	0.01
5	0.05	0.08	0.49	0.02	0.00	0.01
10	0.04	0.12	0.36	0.00	0.01	0.01
20	0.01	0.13	0.34	0.05	0.02	0.01
50	0.10	0.33	0.38	0.03	0.00	0.01

$m = 1$

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.99	0.99	1.00	0.99	1.00	0.99
2	0.99	1.00	0.99	1.00	1.00	0.98
5	0.99	1.00	1.00	1.00	1.00	0.98
10	0.98	1.00	0.99	1.00	1.00	0.99
20	0.99	0.99	1.00	1.00	1.00	0.99
50	0.99	0.99	1.00	1.00	0.98	0.95

$m = 5$

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.99	0.99	1.00	0.99	1.00	0.99
2	0.99	1.00	0.99	1.00	1.00	0.99
5	0.99	1.00	1.00	1.00	1.00	0.98
10	0.98	1.00	0.99	1.00	1.00	0.99
20	0.99	0.99	1.00	1.00	1.00	0.99
50	0.99	0.99	1.00	1.00	0.99	0.95

$m = 10$

Table 4.1: Success probabilities per byte for AES, for various attack parameters.

other error messages are produced.

Packets that pass the test have their `length` set in line 564, a positive value in this case, before the padding is checked in lines 569 to 574. The length of the padding check loop, which translates to the amount of time spent in checking the padding, depends on the value of the padding length byte, which is set every time to $R[b-1] \oplus D_k(C_t^*)[b-1]$, where R is an arbitrary block and C_t^* is the target ciphertext block. The value of the padding length byte corresponds to `pad` in line 550. Clearly, more iterations of the padding check loop (translating to more time) are executed as the value of `pad` increases, introducing a timing side channel that could potentially be exploited in an attack. In addition, the MAC verification performed in lines 582 to 593 is done on data

l n	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.00	0.12	0.39	0.06	0.01	0.13
2	0.03	0.12	0.26	0.03	0.03	0.18
5	0.03	0.20	0.23	0.30	0.07	0.02
10	0.04	0.17	0.09	0.38	0.08	0.04
20	0.14	0.10	0.08	0.22	0.09	0.07
50	0.04	0.08	0.17	0.41	0.15	0.05

$m = 1$

l n	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.99	1.00	1.00	1.00	1.00	0.93
2	0.99	1.00	0.99	0.99	0.93	0.92
5	0.99	1.00	1.00	0.90	0.93	0.83
10	0.99	1.00	0.92	0.89	0.81	0.57
20	0.97	1.00	0.91	0.92	0.77	0.54
50	0.98	1.00	0.90	0.89	0.68	0.59

$m = 5$

l n	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.99	1.00	1.00	0.99	1.00	0.93
2	0.99	1.00	1.00	0.99	0.93	0.92
5	0.99	1.00	1.00	0.91	0.94	0.83
10	0.98	1.00	0.93	0.89	0.81	0.57
20	0.98	1.00	0.92	0.92	0.77	0.54
50	0.99	1.00	0.91	0.89	0.68	0.59

$m = 10$

Table 4.2: Success probabilities per byte for 3DES, for various attack parameters.

with length set in line 564. The time it takes to process packets, of the same length, relies on the number of MAC *compression* function evaluations performed on the data, after removing the padding bytes. Recall that the length of the data, `length`, is set in line 564 and relies on the value of `pad`. *Simplistically*, the time taken by the MAC verification decreases as the value of `pad` increases. Increasing `pad` has the opposite effect on the time it takes to perform the padding check. It is worth noting that the MAC verification takes much more time than the padding check so that the overall effect is to decrease the decryption time as the value of `pad` increases. In summary, the time it takes for the padding check and the MAC verification varies when processing packets of the same size, even when they pass the test. This is reflected in the red

n	p
1	0.017
2	0.210
5	0.205
10	0.012
20	0.035
50	0.147

$m = 1$

n	p
1	0.961
2	0.983
5	0.983
10	0.985
20	0.989
50	0.965

$m = 5$

n	p
1	0.983
2	0.996
5	0.995
10	0.994
20	0.995
50	0.973

$m = 10$

Table 4.3: Success probabilities per byte for AES-256, for $l = 192$, based on 1000 trials.

PDF shown in Figure 4.14 (for packets that pass the test), where we see a flatter distribution, when compared to the blue PDF in the same figure (for packets that fail the test). In Chapter 5, we give more detail on the exact construction of HMACs and describe *similar* timing side channels that we successfully exploited to build further attacks against TLS and DTLS.

The timing difference that arises from packets that fail the test and packets that pass the test allows a partial plaintext recovery attack against GnuTLS. We explain this next. For ease of presentation, we assume that the MAC size is 32 bytes (as would be produced by HMAC-SHA-256), but a similar attack would apply for 20-byte MACs. Now the padding length field `pad` is obtained from the last byte of the decrypted ciphertext (see line 550 in Figure 4.13). Consider an adversary who builds a DTLS packet whose encrypted payload (excluding the IV) is 160 bytes in length and ends with two blocks R, C_t^*. Then, recalling our numbering convention for the bytes of a block and the CBC-mode decryption procedure, the padding length test in the GnuTLS code will fail precisely when:

$$R[b-1] \oplus D_k(C_t^*)[b-1] > 127.$$

Thus, if the targeted system responds quickly to the adversary's packet, he can infer that the most significant bit (MSB) of $R[b-1] \oplus D_k(C_t^*)[b-1]$ is set to 1. From this, the MSB of $P_t^*[b-1]$, the rightmost byte of the plaintext corresponding to C_t^*, is easily deduced, using the standard approach involving the CBC-mode decryption equation, $C_{t-1}^* \oplus D_K(C_t^*) = P_t^*$. The attacker can then target the second-MSB of $P_t^*[b-1]$, by setting $R[b-1]$ so that the MSB of $R[b-1] \oplus D_k(C_t^*)[b-1]$ equals 0 and then using a DTLS packet of length 96 bytes (again excluding the IV). This provides a test of the form:

$$R[b-1] \oplus D_k(C_t^*)[b-1] > 63,$$

```
pad = ciphertext.data[ciphertext.size - 1] + 1;   /* pad */

 if ((int) pad > (int) ciphertext.size - tag_size)
   {
................
    pad_failed = GNUTLS_E_DECRYPTION_FAILED;
   }

    length = ciphertext.size - tag_size - pad;
................
  if (ver != GNUTLS_SSL3 && pad_failed == 0)
   for (i = 2; i < pad; i++)
    {
      if (ciphertext.data[ciphertext.size - i] !=
          ciphertext.data[ciphertext.size - 1])
        pad_failed = GNUTLS_E_DECRYPTION_FAILED;
    }

if (length < 0)
  length = 0;
................
preamble_size =
  make_preamble (UINT64DATA(*sequence), type,
                 length, ver, preamble);
_gnutls_auth_cipher_add_auth (&params->read.cipher_state, preamble,
   preamble_size);
_gnutls_auth_cipher_add_auth (&params->read.cipher_state, ciphertext.data,
   length);
................
ret = _gnutls_auth_cipher_tag(&params->read.cipher_state, tag, tag_size);
```

Figure 4.13: Snapshot from GnuTLS code (gnutls_cipher.c), version 3.0.0.

with the side information that $R[b-1] \oplus D_k(C_t^*)[b-1] \leq 127$, from which the adversary learns the second-MSB of $R[b-1] \oplus D_k(C_t^*)[b-1]$. An alternative approach to this is setting $R[b-1]$ so that the MSB of $R[b-1] \oplus D_k(C_t^*)[b-1]$ equals 1 instead of 0 and then using a DTLS packet of length 224 bytes (again excluding the IV). This provides a test of the form:

$$R[b-1] \oplus D_k(C_t^*)[b-1] > 191,$$

which again allows the adversary to learn the second-MSB of $R[b-1] \oplus D_k(C_t^*)[b-1]$. This alternative approach gives the adversary the opportunity to use packets with sizes that result in better success probabilities, and hence is preferable. For both approaches, iterating, the attacker can extract the 4 MSBs of $P_t^*[b-1]$ when the block cipher is AES, and the 5 MSBs of $P_t^*[b-1]$ when it is 3DES. The least significant bits (LSBs) cannot be extracted using our attack because the packet size must be a multiple of the block size b.

This provides a theoretical description of our attack. Of course, the adversary can use the same techniques as worked for OpenSSL to amplify his attack: using packet trains, multiple trials, and removal of outliers. A practical issue arises because GnuTLS does not implement the Heartbeat extension, but here we can use any application layer protocol with predictable timing differences. In principal, the same attack would work against the GnuTLS implementation of TLS, with the TLS connection tear-down giving the required timing information. But, in this case, trains of packets cannot be used to amplify the timing difference, since the connection is terminated upon the first failure.

We have conducted experiments to test whether the timing difference is sufficient to allow the attack for DTLS, with experimental results being presented in Figure 4.14 for HMAC-SHA-256 and AES-256. Here, we see the slight separation between the two distributions (red for packets where the inequality "`pad > ciphertext.size - hash_size`" is satisfied, blue for when it is not). In this attack, the adversary needs to adjust the payload length, l, based on the position of the bit he tries to recover. Changing the value of l to recover a bit would change the success probability of the attack.

With the second approach, where longer packets are used, we were able to achieve success probabilities of 0.738, 0.744, 0.737 and 0.756 for individually extracting the first, second, third and fourth MSB, respectively, meaning that the four MSBs can be recovered correctly with probability 0.306, using (roughly) 43000 bytes of network traffic. These probabilities were achieved with $n = 5$, $m = 10$ and measured over 1000 attack runs. We used percentile filters, similar to the approach used in [28] to achieve these probability values. As expected, increasing the value of m significantly increases the success probability. For example, we were able to achieve success probabilities of 0.797 and 0.990 for recovering the four MSBs when $m = 50$ and $m = 100$ respectively.

To implement the tests, we used the same hardware set-up as the one we used for OpenSSL. We ran version 3.0.0 of GnuTLS on the client and the server. We used the built-in GnuTLS utilities for the client and the server, `gnutls_cli` and `gnutls_serv` respectively. We again disabled anti-replay by directly modifying the source code. We made use of the *echo* and *echo reply* messages that `gnutls_cli` and `gnutls_serv` implement, in order to compensate the case of not having Heartbeat messages available. Heartbeat requests are replaced by echo messages, while Heartbeat replies are replaced by echo reply messages. This proves the point we made earlier in the chapter which

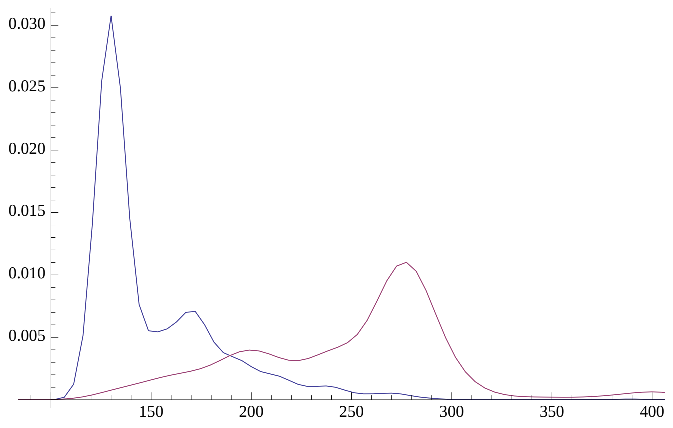


Figure 4.14: PDFs for AES-256 with HMAC-SHA256, $l = 176$ (packet size), $n = 5$ (train size), based on 1000 trials, with outliers removed.

is that the adversary can exploit any messages that predictably generate detectable responses.

4.6 Disclosure

We informed the OpenSSL development team about our attack and worked with them to test their proposed fix[6], which we found to be effective against the attack. OpenSSL applied the fix to releases starting from 1.0.0f and 0.9.8s. Interestingly, the proposed fix introduced a flaw[7] that could be exploited in a denial of service attack; accordingly, users were asked to upgrade to OpenSSL 1.0.0g or 0.9.8t.

We also shared our findings with the GnuTLS development team. We worked with them to identify the root cause of the timing difference and test the proposed fixed[8]. A fix to prevent our specific attack was incorporated in version 3.0.11 of GnuTLS.

The DTLS and TLS libraries of OpenSSL and GnuTLS are used in a number of software packages (for example, the RedHat and SuSe operating systems), which had to be updated accordingly. CVE-2011-4108[9] identifies a number of software packages that implement OpenSSL or GnuTLS, and which were affected by our attacks.

[6]http://www.openssl.org/news/secadv_20120104.txt
[7]http://www.openssl.org/news/secadv_20120118.txt
[8]http://www.gnutls.org/security.html
[9]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4108

4.7 Chapter Conclusion

We have demonstrated plaintext recovering attacks against the OpenSSL and GnuTLS implementations of DTLS. These are easily prevented by modifying the code so that the receiver's cryptographic processing time is independent of how decryption fails. However, we contend that the attacks are still interesting for a number of reasons.

Firstly, the fix to prevent our OpenSSL attack is already mandated in the specification for TLS 1.1, and is implemented in OpenSSL's implementation of TLS, but not in its implementation of DTLS. Without more insight into the software development processes followed by the OpenSSL project, we can only speculate that the experience about how to securely implement TLS's MEE-TLS-CBC construction was not carried over to the separate DTLS implementation. This, then, may also indicate of a lack of truly expert code review in the OpenSSL project. This is concerning given the prominence and wide application of the OpenSSL code, but also understandable given its volunteer-led effort. By contrast, GnuTLS's implementation has common code for the TLS and DTLS packet decryption procedure, meaning that countermeasures implemented for TLS are immediately carried over to DTLS. However, as we saw, even this was not sufficient to fully protect the GnuTLS implementation against the type of attack developed in this chapter.

A second reason that the obvious and mandated countermeasures were not implemented in OpenSSL may stem from DTLS's lack of error messages, which makes the previous attacks apparently impossible against DTLS. We proved otherwise, exploiting DTLS Heartbeat request and response messages to obtain the required timing information. This kind of approach may be more widely applicable than DTLS.

A third possible explanation is that the DTLS specification relies heavily on cross-references to the TLS 1.1 specification, and indeed only gives specification details at points where TLS and DTLS differ. So an implementor needs to be familiar with both specifications in order to implement DTLS properly. We suggest that "specification by `diff`" is not a good approach to specifying secure protocols, since it requires an implementor to jump back and forth between specifications and may allow important details to fall into the gap between.

Secondly, a comparison between our attacks on DTLS and previous attacks on TLS is instructive. Our attacks are in some sense more challenging because of the lack of explicit error messages, but also easier to carry out because of DTLS's tolerance of errors, meaning that DTLS connections are not torn-down whenever an error is encountered as they are in TLS. Ultimately, this error-tolerance comes from DTLS's use of an unreliable transport protocol. For similar reasons, the anti-replay feature in DTLS is made optional in the specification. In this context, our work shows how non-security features of lower layer protocols can have a major influence on security at higher layers. This phenomenon is seemingly not that well-explored in the literature, presenting an interesting challenge for future work.

Our work on DTLS, and in particular our findings in the GnuTL implementation of TLS and DTLS, led us to suspect that there may be further possibilities to exploit the manner in which TLS implementations, following the TLS specification, attempt to achieve constant time decryption processing. Indeed, in the next chapter we present a number of perviously unknown weaknesses in the TLS specification when using the MEE-TLS-CBC construction. These TLS design weaknesses were carried over, as we expected, to DTLS. We also present new techniques to attack TLS and DTLS, exploiting these new weaknesses, and report experimental results that demonstrate the effectiveness of our attacks.

Chapter 5

Lucky Thirteen

5.1 Introduction

In Chapter 3, we discussed the structure of TLS and the various protocols that make it. In Chapter 4, we described a number of attacks against the OpenSSL and GnuTLS implementations of DTLS. In this chapter, we present a family of attacks that apply to the MAC-then-PAD-then-Encrypt construction in all TLS and DTLS implementations that are compliant with TLS 1.1 or 1.2, or with DTLS 1.0 or 1.2. They also apply to implementations of SSL 3.0 and TLS 1.0 that incorporate padding oracle attack countermeasures (implementations that do not are of course already vulnerable to known attacks). Following Chapter 3, we refer to the (D)TLS MEE construction as MEE-TLS-CBC. The core encryption process is illustrated in Figure 5.1 and explained in more detail in Chapter 3 and this chapter.

The attacks come in various distinguishing, partial plaintext recovery, and full plaintext recovery flavours. For the plaintext recovery attacks, no chosen-plaintext capability

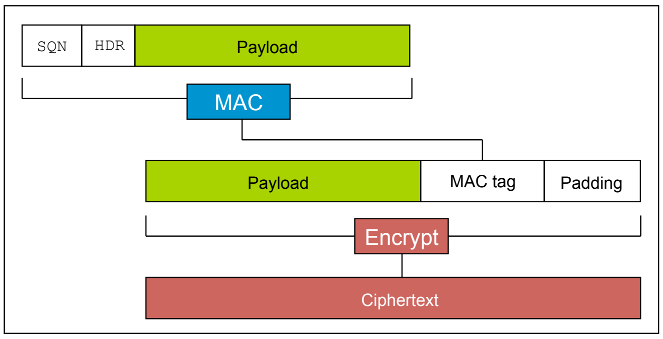


Figure 5.1: D(TLS) encryption process. The figure shows how CBC-mode encryption is applied in TLS and which we refer to as MEE-TLS-CBC in this chapter.

is needed, in contrast to the BEAST attack: the attacks can be mounted by a standard man-in-the-middle (MITM) attacker who sees only ciphertext and can inject ciphertexts of his own composition into the network. The details of which specific attacks are possible depends on the exact size of MAC tags output by the MAC algorithm negotiated by the Handshake Protocol, and also on the fact that exactly 13 bytes of header data are incorporated in the MAC calculation (hence the title of the chapter).

Although the attacks we describe in this chapter make use of the padding oracle, they differ from the ones described earlier in Chapter 4 in various ways:

- The attacks in this chapter exploit weaknesses in the design of TLS and DTLS, as defined in the RFCs, and apply to all versions of TLS and DTLS.
- Our attacks apply to all implementations that follow the TLS and DTLS standards.
- We also developed versions of the attacks that apply to implementations that do *not* follow the TLS and DTLS standards, for example GnuTLS.
- We developed a new realisation of the padding oracle, in which we exploit a smaller timing difference, when compared to the attacks in Chapter 4.

The applicability of the attacks is also implementation-dependent, because of the manner in which different implementations interpret the RFCs. We have investigated several different open-source implementations of TLS and DTLS, and found all of them to be vulnerable to our new attacks or variants of them (or even old attacks in one case). We also found basic coding errors in the security-critical decryption function of one popular implementation, GnuTLS. In view of the amount of variation we have seen in open-source code and our success in devising variant attacks, we expect *all* implementations – whether open or closed – to be vulnerable to our attacks to some extent.

We have implemented a selection of the attacks in an experimental setting. As with earlier attacks, discussed in Chapter 3, completely breaking TLS is challenging because the attacks create “broken” TLS records and so consume many TLS sessions. Nevertheless, our basic attack can extract full plaintext for the current OpenSSL implementation of TLS assuming the attacker is located, say, in the same LAN segment as the targeted TLS client or server, using roughly 2^{23} TLS sessions to reliably recover a block of plaintext in a multi-session attack scenario like that considered in [25]. Such a scenario is applicable when, for example, an application protocol performs automatic TLS reconnection and password retransmission.

Given its complexity, this basic attack would seem to present only a theoretical threat. However, variants of it are much more effective:

- The distinguishing attacks against TLS are quite practical for OpenSSL, requiring just a handful of sessions in order to reliably tell apart the encryptions of chosen messages. We describe the attack in Section 5.3.

- Breaking DTLS implementations is fully practical even for a remote attacker, since we can exploit the fact that DTLS errors are non-fatal to mount the attacks in a single session, and reuse the amplification techniques from Chapter 4 to boost the delicate timing signals on which our attacks depend.

- We also have more efficient *partial* plaintext recovery attacks on TLS and DTLS. For example, against OpenSSL TLS, an attacker who knows one byte of a block in either of the last two byte positions can reliably recover each of the remaining bytes in that block using 2^{16} sessions.

- The complexity of all our attacks can be reduced using language models and sequential statistical techniques as in [25, 35]. As a simple example, if the plaintext is base64 encoded [48], as is the case for HTTP basic access authentication and cookies [17], then the number of TLS sessions needed to recover a block reduces from roughly 2^{23} to 2^{19}.

- In the web setting, our techniques can be combined with those used in the BEAST attack [35], discussed in Section 3.10: a JavaScript (or similar) running in the browser can be used to initiate all the needed TLS sessions, with an HTTP cookie being automatically injected by the browser in a predictable location in the plaintext stream in each session. The JavaScript can also control the location of the cookie such that there is only one unknown byte in the target block at each stage of the attack. The attacker then combines the "one known byte" variant of our attack and the base64 optimisation above (assuming the sensitive part of the cookie is base64 encoded). Putting all of these improvements together, we estimate that HTTP cookies can be recovered using 2^{13} sessions per byte of cookie (with all the sessions being automatically generated by the JavaScript and the browser). Note that the JavaScript does not need the ability to inject chosen plaintext into an existing TLS session for our attack to work.

Our new attacks exploit the fact that, when badly formatted padding is encountered during decryption, a MAC check must still be performed on *some* data to prevent the known timing attacks discussed in Chapter 3. But what data should be used for that calculation? The TLS 1.1 and 1.2 RFCs recommend checking the MAC as if there was a zero-length pad. As noted in those RFCs:

> *This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.*

We confirm that there are indeed small timing differences, but, contrary to what is written in the RFCs, they *can* be exploited. In short, provided there is a fortuitous alignment of various factors such as the size of MAC tags, the block cipher's block size, and the number of header bytes, then there will be a time difference in the time that it takes to process TLS records having good and bad padding, and this difference will show up in the time at which error messages appear on the network. This timing side-channel can then be "wrangled" into revealing plaintext data via careful statistical analysis of multiple timing samples. As we shall show, other natural methods for handling MAC checking in the event of bad padding also lead to exploitable timing differences.

Our new attacks demonstrate that properly implementing MAC-then-PAD-then-Encrypt in (D)TLS so as to avoid all exploitable timing differences is in fact quite difficult, and is not achieved by any of the implementations we examined. A complicating factor, in addition to dealing with padding, is the need for careful sanity checking of various fields during decryption. We provide a detailed prescription for dealing with these issues. We also discuss other, more easily-implemented countermeasures.

It is worth noting that recent work on the security of TLS implementations includes [36, 40, 8]. In particular, in independent work, Pironti *et al.* [8] identify effectively the same timing channel in TLS that we exploit. However they dismiss it as being "too small to be measured over the network" and instead focus on using it to recover information about message lengths.

Section 5.2 provides further background on the HMAC calculation, which helps to establish an understanding of the fundamental root cause of the timing difference we exploit in the attacks. Section 5.3 presents the basic distinguishing attack against RFC-compliant implementations of TLS and DTLS, while Section 5.4 describes our plaintext recovery attacks in the context of TLS and explains how to modify them to apply to DTLS. In Section 5.5 we report on the experimental validation of our attacks for the OpenSSL implementation, and in Section 5.6 we describe the modifications needed to make our attacks applicable to other implementations, including GnuTLS, CyaSSL and PolarSSL. Section 5.7 discusses countermeasures to our attacks, giving guidance on how to implement MEE-TLS-CBC so as to avoid the attacks. It also includes details of how we disclosed our attacks and how vendors reacted to them. Finally, Section 5.8 concludes with a recap of the main issues raised by our work in this chapter.

5.2 Details of HMAC

In Section 3.3.2, we discussed a number of constructions for calculating MACs. TLS and DTLS exclusively use the HMAC algorithm [58], with HMAC-MD5, HMAC-SHA-1, and HMAC-SHA-256 being supported in TLS 1.2[1]. To compute the MAC tag T for a message M with key K_a, HMAC applies the specified hash algorithm H twice, in an iterated fashion:

$$T = H((K_a \oplus \texttt{opad}) \mid\mid H((K_a \oplus \texttt{ipad}) \mid\mid M)). \tag{5.1}$$

Here `opad` and `ipad` are specific 64-byte values, and the key K_a is zero-padded to bring it up to 64 bytes before the XOR operations are performed. For all the hash functions H used in TLS, the application of H itself uses an encoding step called *Merkle-Damgård strengthening*. Here, an 8-byte length field followed by padding of a specified byte format are appended to the message M to be hashed. The padding is at least 1 byte in length and aligns the data on a 64-byte boundary. The relevant hash functions also have an iterated structure, processing messages in chunks of 64 bytes (512 bits) using a compression function, with the output of each compression step being chained into the next step. The compression function in turn involves a complex round structure, with many basic arithmetic operations on data being involved in each round.

In combination, these features mean that HMAC implementations for MD5, SHA-1 and SHA-256 have a distinctive timing profile. Messages M of length up to 55 bytes can be encoded into a single 64-byte block, meaning that the first, inner hash operation in HMAC is done in 2 compression function evaluations (since its input is a 128-byte string), with 2 more being required for the outer hash operation, for a total of 4 compression function evaluations. Messages M containing from 56 up to $64+55 = 119$ bytes can be encoded in two 64-byte blocks, meaning that the inner hash is done in 3 compression function evaluations, with 2 more being required for the outer operation, for a total of 5. In general, an extra compression function evaluation is needed for each additional 64 bytes of message data, with the exact number needed being given by the formula $\lceil \frac{\ell-55}{64} \rceil + 4$, where ℓ is the message length in bytes. A single compression function evaluation takes typically around 500 to 1000 hardware cycles (depending on

[1]TLS cipher suites using HMAC with SHA-384 are specified in RFC 5289 (ECC cipher suites for SHA256/SHA384) and RFC 5487 (Pre-Shared Keys SHA384/AES) but we do not consider this version of HMAC further here.

the hash function and details of the implementation), giving a time in the sub-μs range for modern processors.

Recall that in TLS the MAC is computed on plaintext *after* removing padding. Hence, one might expect the total running time for decryption processing to reveal some information about the *size* of the depadded plaintext, perhaps up to a resolution of 64 bytes in view of the above discussion. Our distinguishing attack exploits this, but we will show that much more is possible.

5.3 A Distinguishing Attack

In this section we describe a simple distinguishing attack against the MEE-TLS-CBC construction as used in TLS. This is a warm-up to our plaintext recovery attacks, but we note that even a distinguishing attack against such an important protocol would usually be regarded as a significant weakness.

In a distinguishing attack, the attacker gets to choose pairs of messages (M_0, M_1). One of these is encrypted, M_d, say, and the resulting ciphertext is given to the attacker. The attacker's task is to decide the value of the bit d. To prevent the attacker from winning trivially, we require that M_0 and M_1 have the same length.

To visualise (and simplify) the configuration, the reader can think of an attacker submitting (M_0, M_1) to a web server that would return M_d. The attacker can choose a reliable method when submitting the pairs of messages to the web server. Again, the attacker's task is to decide the value of the bit d.

We focus on the case where $b = 16$, i.e. the block cipher is AES. A variant of the attack works for $b = 8$. Suppose the MAC algorithm is HMAC-H where H is either MD5, SHA-1 or SHA-256. Let M_0 consist of 32 arbitrary bytes followed by 256 copies of 0xFF. Let M_1 consist of 287 arbitrary bytes followed by 0x00. Note that both messages have 288 bytes, and hence fit exactly into 18 plaintext blocks. Our attacker submits the pair (M_0, M_1) for encryption and receives a MEE-TLS-CBC ciphertext HDR $||$ C. Now C consists of a CBC-mode encryption of an encoded version of M_d, where the encoding step adds a MAC tag T and some padding pad. Because the end of M_d aligns with a block boundary, the additional bytes T $||$ pad are encrypted in separate blocks from M_d. The attacker now forms a new ciphertext HDR $||$ C' in which C' keeps the same 16-byte IV as C (if explicit IVs are being used), but truncates the non-IV part of C to 288 bytes. This has the effect of removing those blocks of C that contain T $||$ pad.

Now the attacker submits HDR $||$ C' for decryption. If the record underlying C was M_0, then the plaintext P' corresponding to C' appears to end with the valid 256-byte padding pattern 0xFF...0xFF. In this case, all of these bytes are removed, and the remaining 32 bytes of plaintext are interpreted as a short message and a MAC tag. For example, if H is SHA-1, then we have a 12-byte message and a 20-byte MAC tag. The MAC verification fails (with overwhelming probability), and an error message is returned to the attacker. If the underlying record was M_1, then P' appears to end with the valid 1-byte padding pattern 0x00. In this case, a single byte is removed, and the remaining 287 bytes of plaintext are interpreted as a long message and a MAC tag. Again, the MAC verification fails and an error message is returned to the attacker.

Notice that when $d = 0$, so C encrypts M_0, a short message consisting of 13 bytes of header plus at most 16 bytes of message (when the hash algorithm is MD5) is passed through the MAC algorithm. To calculate the MAC requires 4 evaluations of H's compression function. On the other hand, when $d = 1$, C encrypts M_1, and a long message consisting of 13 bytes of header plus at least 255 bytes of message is passed through the MAC algorithm. Then to calculate the MAC requires at least 8 evaluations of H's compression function, at least 4 more than for the $d = 0$ case. Hence, we expect the time it takes to produce the error message on decryption failure to be somewhat larger if $d = 1$ than when $d = 0$, on the order of a couple of μs for a modern processor. This timing difference then allows, in theory, a distinguishing attack on the MEE-TLS-CBC construction used in TLS.

5.3.1 Practical Considerations

In describing the attack, we have ignored the time taken to remove padding. This is different for the two messages being processed, and the difference is opposite to that for MAC checking in that padding removal for M_0 takes longer than for M_1. Similarly, we have ignored any other timing differences that might arise during other processing steps. In practice, as we will see in Section 5.5, these differences turn out to be smaller than the MAC timing difference.

The attack exploits the requirement from the (D)TLS RFCs that implementations be able to properly decrypt records having variable length padding, but does not require implementations to actually send records containing such padding. A variant attack is possible in case only minimum-length padding is supported, but involves a smaller timing signal.

In TLS, the error messages are sent over the network, and so can easily be detected by the attacker. However, these messages are subject to network jitter, and this may be large enough to swamp the timing difference arising from the 4 extra compression function evaluations. On the other hand, the timing signal may be quite large when the cryptographic processing is performed in a constrained environment, e.g. on an 8-bit or 16-bit processor, or even on a smartphone. Furthermore, the jitter may be significantly reduced when the adversary runs as a separate process on the machine performing TLS decryption. This may be possible in virtualised environments, e.g. in a cloud scenario as explored in [86]. The attack also destroys the TLS session, since in TLS such errors are fatal. The attack can be iterated across L sessions, with M_d being encrypted in each session, and statistical processing used to extract the timing signal.

In DTLS, there are no error messages, but the techniques from Chapter 4 can be applied to solve this problem. There, we send a packet containing a ciphertext C closely followed by a DTLS message, with the latter always provoking a response message. Any timing difference arising from the decryption of C then shows up as a difference in the arrival time of the response messages. The signal amplification techniques from Chapter 4 can also be used to boost the timing difference – here, the idea is to send multiple packets all containing C in quick succession, to create a cumulative timing difference (since each time C is processed, it will be processed in the same way).

In the attack as described, we have used 288 byte messages. This ensured that there were sufficient bytes left after the removal of padding to leave room for a message (possibly of zero length) and a MAC tag. This ensures that C' passes any sanity checks that might be applied during decryption. However, these sanity checks might be exploitable in variants of our basic attack. For example, an implementation that finds it does not have enough bytes left to contain a MAC after depadding may choose to skip MAC verification altogether, leading to an increased timing difference. In fact, we saw such behaviour in Chapter 4 for the GnuTLS implementation.

Note that the attack would still work as described if the truncated MACs specified for TLS in [2] were used, since the full HMAC-H computation is still performed but only certain bytes of the computed tag are compared to bytes of the plaintext.

We report on the successful implementation of this attack in Section 5.5.

5.4 Plaintext Recovery Attacks

5.4.1 General Approach

As we have seen in the previous section, the processing time for a (D)TLS record (and therefore the appearance time of error messages) will depend on the amount of padding that the receiver interprets the encoded plaintext as containing. However, by placing a target ciphertext block at the end of the encrypted record, an attacker can arrange that the plaintext block corresponding to this block is interpreted as padding, and hence make the processing time depend on plaintext bytes. But, it seems that large amounts of valid padding are needed to create a significant timing difference, and this is difficult to arrange in a plaintext recovery attack. We show that this barrier to plaintext recovery can be overcome under certain circumstances.

Let C^* be any ciphertext block whose corresponding plaintext P^* the attacker wishes to recover. Let C' denote the ciphertext block preceding C^*. Note that C' may be the IV or the last block of the preceding ciphertext if C^* is the first block of a ciphertext. We have:

$$P^* = D_{K_d}(C^*) \oplus C'.$$

Following Chapter 3, for any block B made of b bytes of plaintext or ciphertext, we write $B = [B[0]B[1] \dots B[b-1]]$, where $B[i]$ denote the bytes of B. In particular, we have $P^* = [P^*[0]P^*[1] \dots P^*[b-1]]$.

As usual, we assume that the attacker is capable of eavesdropping on the (D)TLS-protected communications and of injecting messages of his choice into the network. For TLS, or DTLS with sequence number checking disabled, we do not need the ability to prevent messages from reaching their destination, nor do we require a chosen-plaintext capability.

5.4.2 Full Plaintext Recovery

For simplicity of presentation, in what follows, we assume the CBC-mode IVs are explicit (as in TLS 1.1, 1.2 and DTLS 1.0, 1.2, described in Section 3.4.7). We also assume that $b = 16$ (so our block cipher is AES). It is easy to construct variants of our attacks for implicit IVs and for $b = 8$. We begin by considering only TLS, with details for DTLS to follow. We also assume that the TLS implementation follows the advice in the TLS 1.1 and 1.2 RFCs about checking the MAC as if there was a zero-length pad when the padding is incorrectly formatted. We will examine the security of other implementation options in Section 5.6. Most importantly, and for reasons that will become clear, we assume for the moment that $t = 20$ (so that the MAC algorithm is HMAC-SHA-1). We consider $t = 16$ and $t = 32$ (HMAC-MD5 and HMAC-SHA-256) shortly.

Let Δ be a block of 16 bytes and consider the decryption of a ciphertext $C^{\text{att}}(\Delta)$ of the form

$$C^{\text{att}}(\Delta) = \texttt{HDR} \,||\, C_0 \,||\, C_1 \,||\, C_2 \,||\, C' \oplus \Delta \,||\, C^*$$

in which there are 4 non-IV ciphertext blocks (blocks not containing an IV), the penultimate block $C' \oplus \Delta$ is an XOR-masked version of C' and the last block is C^*. The corresponding 64-byte plaintext is $P = P_1 \,||\, P_2 \,||\, P_3 \,||\, P_4$ in which

$$\begin{aligned} P_4 &= D_{K_e}(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Notice that P_4 is closely related to the unknown, target plaintext block P^*. We consider 3 distinct cases, which between them cover all possibilities for what can happen during decryption of $C^{\mathrm{att}}(\Delta)$:

1. P_4 ends with a 0x00 byte: in this case, a single byte of padding is removed, the previous 20 bytes are interpreted as a MAC tag T, and the remaining $64-21 = 43$ bytes of plaintext are taken as the record R. MAC verification is then performed on a $13 + 43 = 56$-byte message `SQN` $||$ `HDR` $||$ R. Recall, exactly 13 bytes of data are prepended to the record R here before the MAC is computed (the 8-byte sequence number and the 5-byte header).

2. P_4 ends with a valid padding pattern of length at least 2 bytes: in this case, at least 2 bytes of padding are removed, and the next 20 bytes are interpreted as a MAC tag T. This leaves a record R of length at most 42 bytes, meaning that MAC verification is then performed on a message of length at most 55 bytes.

3. P_4 ends with any other byte pattern: in this case, the byte pattern does not correspond to valid padding. Following the prescription in the TLS 1.1 and 1.2 RFCs, the plaintext is treated as if it contains no bytes of padding, so the last 20 bytes are interpreted as a MAC tag T, and the remaining 44 bytes of plaintext are taken as the record R. MAC verification is then performed on a 57-byte message.

In all cases, the MAC verification will fail (with overwhelming probability) and an error message produced. Notice that, in accordance with the discussion in Section 5.2, in Cases 1 and 3, the MAC verification will involve 5 evaluations of the compression function for SHA-1, while Case 2 only requires 4 evaluations. Therefore, we can hope to distinguish Case 2 from Cases 1 and 3 by timing the appearance of the error message on the network. Here the timing difference is that needed for a single SHA-1 compression function evaluation (compared to 4 such evaluations in our distinguishing attack). Notice that the size of the header, 13 bytes, in conjunction with the MAC tag size, 20 bytes, are critical in generating this distinctive timing behaviour.

In Case 2, assuming that the plaintext has no special structure, the most likely padding pattern to arise is the one of length 2, namely 0x01 $||$ 0x01, with all longer padding patterns being roughly 256 times less likely. Thus, if the attacker selects a mask Δ in such a way that he detects Case 2 after submitting $C^{\mathrm{att}}(\Delta)$ for decryption, then he can infer that P_4 ends with 0x01 $||$ 0x01, and, using the equation $P_4 = P^* \oplus \Delta$, can now recover the last 2 bytes of P^*. (In fact, by repeating the attack with a mask Δ' that is modified from Δ in the third-to-last byte, the attacker can easily separate the case of a length 2 padding pattern from all longer patterns.)

The question remains: how does the attacker trigger Case 2, so that he can extract the last 2 bytes of P^*? Recall that the attacker has the freedom to select Δ. By injecting a sequence of ciphertexts $C^{\mathrm{att}}(\Delta)$ with values of Δ that vary over all possible values in the last 2 bytes $\Delta[14], \Delta[15]$, then (in the worst case) after 2^{16} trials, the attacker will surely select a value for Δ such that $C^{\mathrm{att}}(\Delta)$ triggers Case 2.

Once the last 2 bytes of P^* have been extracted, the attacker can more efficiently recover the remaining bytes of P^*, working from right to left. This phase is essentially identical to Vaudenay's original padding oracle attack [102] discussed in Chapter 3. For example, to extract the third-to-last byte, the attacker can use his new knowledge of the last two bytes of P^* to now set $\Delta[14], \Delta[15]$ so that P_4 ends with `0x02` || `0x02`. Then he generates candidates $C^{\mathrm{att}}(\Delta)$ as before, but modifying $\Delta[13]$ only. After at most 2^8 trials, he will produce a ciphertext which falls into case 2 again, which reveals he has managed to set a value `0x02` in the third-to-last byte of $P_4 = P^* \oplus \Delta$. From this, he can recover $P^*[13]$. Recovery of each subsequent byte in P^* requires at most 2^8 trials, giving a total of $14 \cdot 2^8$ trials to complete the extraction of P^*.

Practical considerations: In practice, for TLS, there are two severe complications. Firstly, the TLS session is destroyed as soon as the attacker submits his very first attack ciphertext. Secondly, the timing difference between the cases is very small, and so likely to be hidden by network jitter and other sources of timing difference.

The first problem can be overcome for TLS by mounting a multi-session attack, wherein we suppose that the same plaintext is repeated in the same position over many sessions (as in Canvel *et al.* [25], for example). We have used masks Δ in such a way that no further modification to the attack is needed to cater for this setting – of course blocks C' and C^* change for each session.

The second problem can be overcome in the same multi-session setting by iterating the attack many times for each Δ value and then performing statistical processing of the recorded times to estimate which value of Δ is most likely to correspond to Case 2. In practice, we have found that a basic percentile test (and even averaging) works well – see Section 5.5 for further details. Assuming that L trials are used for each Δ value, the attack as described consumes roughly $L \cdot 2^{16}$ sessions, with one ciphertext $C^{\mathrm{att}}(\Delta)$ being tried in each session.

More efficient variants: The attack complexity can be significantly reduced by assuming that the language from which plaintexts are drawn can be modelled using a finite-length Markov chain. This is a fair assumption for natural languages, as well as application-layer protocol messages such as HTML, XML etc. This model can be used to drive the selection of candidate plaintext bytes in order of decreasing likelihood, and from this, determine the bytes of Δ needed to test whether a guess for the plaintext bytes leads to valid padding or not. Similar techniques were used in [25, 35] in combination with sequential statistical techniques to reduce the complexity of recovering low-entropy plaintexts. Note that this approach does not work well if TLS's optional compression is used. Another possibility is that the plaintext bytes are drawn from a reduced space of possibilities. For example, in HTTP basic access authentication, the username and password are Base64 encoded, meaning that each byte of plaintext has only 64 possible values. Similar restrictions often apply to the sensitive parts of HTTP cookies.

In a related attack scenario, if the attacker already knows one of the last two bytes of P^*, he can recover the other byte with much lower complexity than our analysis so far would suggest. This is then a plaintext recovery attack with partially-known-plaintext. For example, suppose the attacker knows the value of the byte $P^*[14]$. Then he sets the starting value of Δ such that $\Delta[14] = P^*[14] \oplus \texttt{0x01}$, so that when $C^{\text{att}}(\Delta)$ is decrypted, the second-to-last byte of P_4 already equals 0x01. Then he iterates over the 2^8 possible values for $\Delta[15]$, eventually finding one such that P_4 has its last two bytes equal to 0x01 $||$ 0x01, triggering Case 2. He can then proceed to recover the rest of P^* with the same complexity as before. Overall, this attack, which recovers 15 bytes of plaintext with 1-out-of-2 of the last bytes of the target block known, consumes only $15L \cdot 2^8$ sessions, where L is the number of trials used for each Δ value in each byte position. This can be further reduced by combining the two variants. For example, for base64 encoded plaintext, only $15L \cdot 2^6$ sessions are needed to decrypt a block.

Combining Lucky 13 with BEAST: A significant limitation of our attacks as described so far is their consumption of many TLS sessions. This limitation can be overcome by combining our attacks with techniques from the BEAST attack [35], discussed in Section 3.10, to target TLS-protected HTTP cookies. The combined attack does not require the blockwise privilege needed by the original BEAST attack (because the attacker does not need to be able to prepend a plaintext block to ongoing HTTP requests). Assuming the targeted part of the cookie is base64 encoded, the attack consumes $L \cdot 2^6$ sessions per byte of HTTP cookie. As we will discuss in more detail in Section 5.5, we found that setting $L = 2^7$ yields reliable plaintext recovery in our experimental set-up, giving us an attack that recovers HTTP cookies using roughly 2^{13} sessions per unknown byte of cookie.

We effectively use the BEAST-style JavaScript means to be in the 1-out-of-2 bytes known case, described above. It is worth noting that our attack does not require the BEAST-style same-origin policy (SOP) bypass. We only require to be able to pad HTTPS requests (for example, `GET` or `POST` requests). In this case, the browser will automatically establish the needed TLS sessions when the JavaScript makes the HTTPS requests.

5.4.3 Plaintext Recovery for Other MAC Algorithms

A critical feature of our attack above is the relationship between the size of the header included in the MAC calculation (fixed at $h = 13$ bytes), the MAC tag size t, and the block size b. For example, if TLS happened to be designed such that $h = 12$, then, with $t = 20$ and $b = 16$, a similar case analysis as before shows that our ciphertext $C^{\mathrm{att}}(\Delta)$ would have the property of having faster MAC verification if P_4 also ends with the single byte `0x00` (the valid padding pattern of length 1). This would allow an improved 2^8 attack against TLS with CBC-mode and HMAC-SHA-1. In some sense, 13 *is* lucky, but 12 would have been luckier!

Similarly, we have (less efficient) variants of our attacks for HMAC-MD5 and HMAC-SHA-256, where the tag sizes t are 16 and 32 bytes, respectively. In fact, because here t is a multiple of b, the analysis is largely the same in both cases, and we consider only HMAC-MD5 in detail. This time $C^{\mathrm{att}}(\Delta)$ is such that we fall into Case 2 (valid padding with a message of size at most 55 bytes, giving fast MAC verification) only if $P_4 = P^* \oplus \Delta$ ends with a valid padding of length 6 or more. With no additional information on P^* the attacker would need (worst case) 2^{48} attempts to construct the correct Δ so as to trigger this case; detecting that he had done so would be more difficult in view of the large number of candidate Δ values. This is not an attractive attack, especially in view of the practical considerations for TLS mentioned above. On the other hand, we do have attractive partially-known-plaintext attacks for HMAC-MD5 and HMAC-SHA-256. For example, if any 5 out of the last 6 bytes of P^* are known, we can recover the remaining 11 bytes using $11L \cdot 2^8$ sessions. The attack can also be made more efficient if the plaintext has low entropy, by trying candidates for the last 6 bytes of P^* in order of decreasing probability and then recovering the remaining bytes of P^* once the right 6-byte candidate is found. This would be an good option for password recovery, for example.

A similar analysis can be carried out for truncated MAC algorithms, as per [2]. For example, for an 80-bit (10-byte) MAC tag, if any 11 out of the last 12 bytes of P^* are known, we can recover the remaining 5 bytes using $5L \cdot 2^8$ sessions.

Finally, we note that the "Lucky 13 + BEAST" attacks work equally well, no matter what the MAC tag size is.

5.4.4 Applying the Attacks to DTLS

So far we have focussed on TLS. The changes needed to handle DTLS are the same as for our distinguishing attack in Section 5.3: we can use the techniques discussed in Chapter 4 to amplify the timing differences and to emulate TLS's error messages. The amplification capability reduces the attack complexity dramatically: essentially, we can accurately test each Δ value using just a few packet trains instead of requiring L trials.

There is one further critical difference that we wish to emphasise: as already noted, DTLS does not treat errors arising during decryption as being fatal. This means that the entire attack against DTLS can be carried out in a *single* session, that is, without requiring the same plaintext to be repeated in the same position in the plaintext across multiple sessions, and without waiting for the Handshake Protocol to rerun for each session.

These differences brings our attack well within the bounds of practicality for DTLS. This is particularly so if DTLS's optional checking of sequence numbers is disabled. Even if this is not the case, the attacks are quite feasible in practice, provided enough DTLS Heartbeat messages are available, or if the upper layer protocol being protected by DTLS produces replies to sent messages in a consistent manner. These points are discussed at greater length in Chapter 4 and the next section, where we report on the successful implementation of our attacks for the OpenSSL implementation of TLS and DTLS.

5.5 Experimental Results for OpenSSL

5.5.1 Experimental Setup

We ran version 1.0.1 of OpenSSL on the client and the server. In our laboratory set-up, a client, the attacker and the targeted server are all connected to the same VLAN on a 100Mbps Ethernet switch. The targeted server was running on a single core processor machine operating at 1.87 GHz with 2 GByte of RAM, while the attacker was running on a dual core processor machine operating at 3.4 GHz, with 2 GByte of RAM.

To simulate the (D)TLS client, we made use of `s_client`, a generic tool that is available as part of the OpenSSL distribution package. We developed a basic Python script that calls `s_client` whenever the TLS connection is terminated, implementing the multi-session setting discussed in Section 5.3.1 of this chapter. Our attack code is written in C and is capable of capturing, manipulating and injecting packets of choice into the network.

In the case of TLS, the attacker captures the "targeted" packet, i.e. the packet containing the plaintext that the attacker tries to recover, manipulates it and then sends the crafted version to the targeted server causing the TLS session to terminate. This crafted packet forces the client and the targeted server to lose TCP synchronisation, causing delay in the TCP connection shutdown. To speed up the TCP connection tear down, the attacker sends spoofed RST packets to the client and the targeted system upon detecting the TLS encrypted alert message, forcing both systems to independently destroy the underlying TCP structure associated with the terminated TLS session.

All the timing values presented in the chapter are based on hardware cycles, which are specific to processor speed. For example, 187 hardware cycles on our targeted server operating at speed of 1.87 GHz translate to an absolute timing of 100ns. To count the hardware cycles, we made use of an existing C library licensed under GNU GPL v3[2].

5.5.2 Statistical Analysis

The network timings we collect in each experiment are from skewed distribution(s) with long tails and many outliers. However, we found that using basic statistical techniques (medians and, more generally, percentiles) was sufficient to analyse our data.

5.5.3 Distinguishing Attack for OpenSSL TLS

Figure 5.2 shows the experimental distribution of timing values for the TLS distinguishing attack described in Section 5.3. The figure indicates that, with enough samples, it should be possible to distinguish encryptions of message M_0 (consisting of 32 arbitrary bytes followed by 256 copies of 0xFF) from encryptions of message M_1 (consisting of 287 arbitrary bytes followed by 0x00).

We used a simple threshold test to build a concrete attack: we calculate a threshold value T based on profiling, gather L timing samples, filter outliers, calculate the median of the remaining timing samples, and then output 1 if the median value is greater than T and 0 if it is less. Table 5.1 shows the success probabilities for this concrete distinguishing attack; it is evident that the attack is reliable even if only a moderate number of samples are available. The attack already has a significant advantage over guessing when $L = 1$, i.e. when only one sample is available.

5.5.4 Plaintext Recovery Attacks for OpenSSL TLS

Partial plaintext recovery: Section 5.4 describes an attack where byte $P^*[15]$ can be recovered when $P^*[14]$ is known. This involves setting $\Delta[14]$ to force $P^*[14] \oplus \Delta[14]$

[2]`http://code.google.com/p/fau-timer`

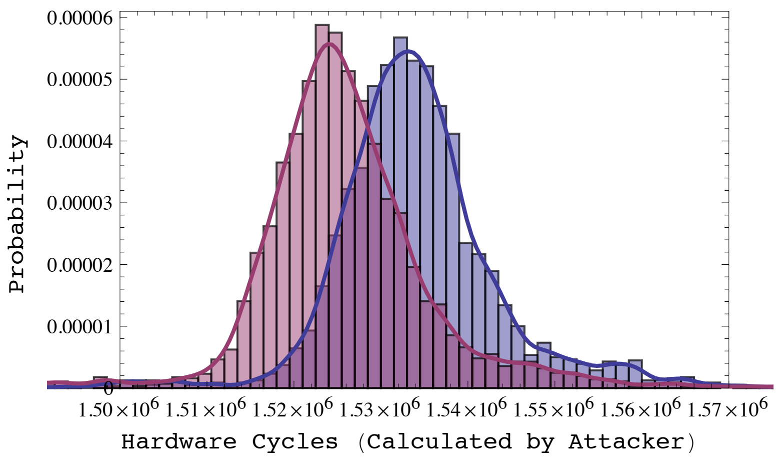


Figure 5.2: Distribution of timing values (outliers removed) for distinguishing attack on OpenSSL TLS, showing faster processing time in the case of M_0 (in red) compared to M_1 (in blue).

L	Success Probability
1	0.756
2	0.769
4	0.858
8	0.914
16	0.951
32	0.983
64	0.992
128	1

Table 5.1: OpenSSL TLS distinguishing attack success probabilities.

to equal 0x01, and then trying all possible values of $\Delta[15]$, identifying which one forces $P^*[15] \oplus \Delta[15]$ to also equal 0x01. Figure 5.3 shows the median server-side decryption time as a function of $\Delta[15]$ for the particular values of $P^*[14] =$ 0x01 (so $\Delta[14] =$ 0x00) and $P^*[15] =$ 0xFF. A clear reduction in processing time can be seen for the expected value of $\Delta[15]$, namely $\Delta[15] =$ 0xFE. Also notable is the stability in the median processing time for other byte values. These server-side times indicate that an attack based on timing error message on the network has some prospect of success. Figure 5.4 shows the corresponding distribution of median network timings in our experimental setup. Clearly, the data is noisier, but the "dip" at $\Delta[15] =$ 0xFE is clearly distinguishable.

Figure 5.5 shows success probabilities for the attack. Each data-point in the figure is based on at least 64 experiments. Each curve in the figure represents a different number of total sessions consumed in the attack (corresponding to different values for

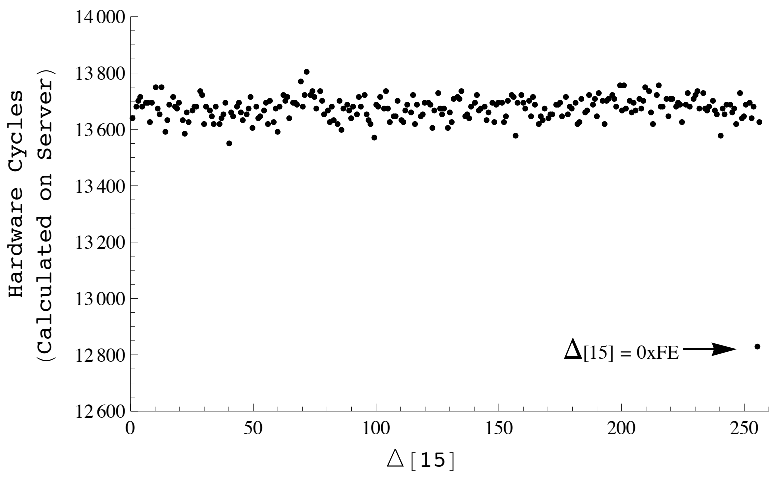


Figure 5.3: OpenSSL TLS median server timings (in hardware cycles) when $P^*[14] =$ 0x01 and $P^*[15] =$ 0xFF. As expected, $\Delta[15] =$ 0xFE leads to faster processing time.

L, the number of trials for each Δ value). The x-axis represents the percentile used in our statistical test[3]: if the percentile value is p, then we take as the correct value for $\Delta[15]$ the one for which the p-th percentile value of the timing distribution (measured over L samples) is *minimised.* It is evident that a range of percentiles work well, including the median. As expected, the success probability of the attack increases as L increases. We already reach a success probability of 1 when $L = 2^8$, where the total number of sessions needed is 2^{16}. Similarly, we have a success probability of 0.93 when $L = 2^7$, where the total number of sessions is 2^{15}.

Given these results, we anticipate that the attack would extend easily to recovering 15 unknown bytes from a block, given one of the last two bytes. We have not implemented this variant.

It is worth noting that the amount of connections required for the attack would potentially trigger alarms on network intrusion detection tools, in case they have been deployed in the path that the attack takes.

Full plaintext recovery: The next step would be to perform the full plaintext recovery attack from Section 5.4. In this case, the attacker would need a total of $L \cdot 2^{16}$ trials to discover which mask value triggers Case 2. In the case of TLS, this takes a considerable amount of time due to the underlying TCP and TLS connection set-up and tear-down times. For example, with $L = 2^7$ we estimate that the 2^{23} sessions would take around 64 hours in our setup. However, once the last two bytes of a block have

[3]We used the GNU statistics package (`http://www.gnu.org/software/gsl/manual/html_node/Statistics.html`) for calculating the percentiles. The algorithm for computing the percentiles involves interpolation, which can over-estimate the success probability for small values of L.

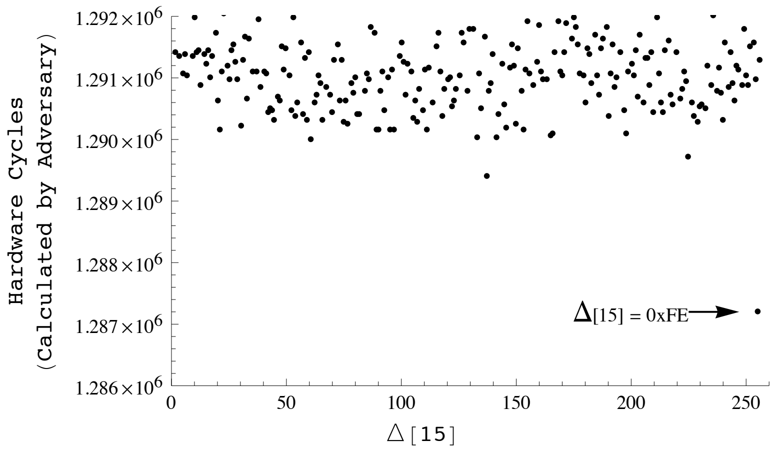


Figure 5.4: OpenSSL TLS median network timings in terms of hardware cycles when $P^*[14] = \texttt{0x01}$ and $P^*[15] = \texttt{0xFF}$. As expected $\Delta[15] = \texttt{0xFE}$ leads to faster processing time.

been successfully recovered, then the remaining bytes in that block can be recovered in a much shorter time. We have not implemented the full plaintext recovery attack for TLS. Our results below for DTLS strongly indicate that the full attack would work for TLS with $L = 2^7$, albeit slowly.

5.5.5 Plaintext Recovery Attacks for OpenSSL DTLS

As explained in Section 5.4.4, we can use the timing and amplification techniques from Chapter 4 in combination with the previously described attacks to attack DTLS. Now the attacker sends a number (n) of crafted packets, followed by a DTLS Heartbeat request and waits for the corresponding Heartbeat reply. This process is repeated L times for each mask value. The attacker selects n and L in order to trade-off the attack success probability and the total number of packets injected. We have found experimentally that $n = 10$ is a good choice for achieving stable timing values. On the other hand, $n = 1$ is indicative of what might be expected to happen with TLS but without enduring the overhead of TCP and TLS connection setups (note that the noise levels for DTLS are generally somewhat higher since we depend on an application-layer error message rather than a native TLS error message). Higher values of n could be used if the attacker is remote from the server.

Figure 5.6 shows the percentile-based success probabilities for recovering $P^*[15]$ assuming that $P^*[14]$ is known, for $n = 10$. It can be seen that the attack is very effective, reliably recovering the unknown plaintext byte with only 2^{11} trials $(L = 2^3)$. Even for 2^8 trials $(L = 1)$, the success probability is 0.266.

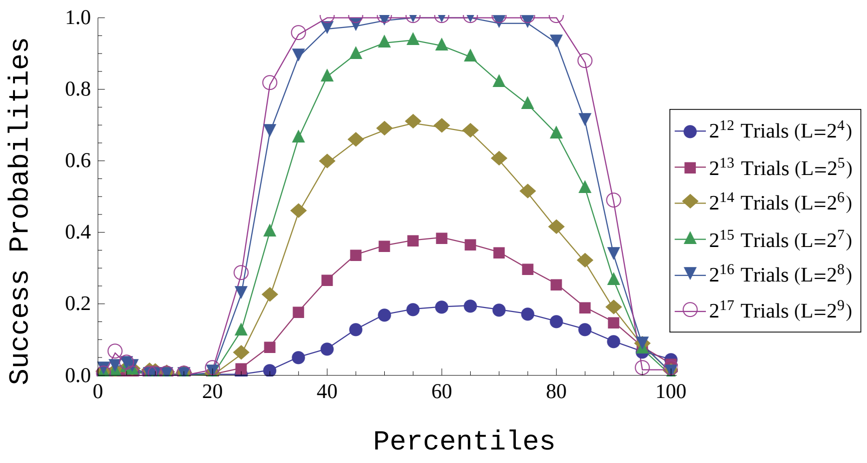


Figure 5.5: OpenSSL TLS partial plaintext recovery: percentile-based success probabilities for recovering $P^*[15]$ assuming $P^*[14]$ is known.

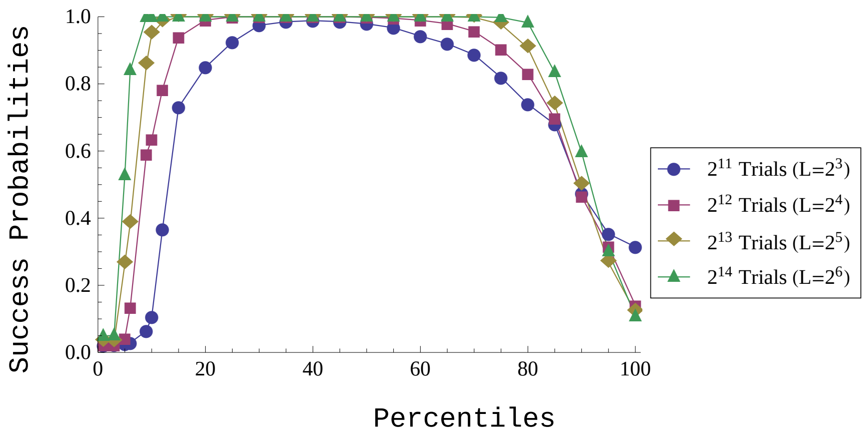


Figure 5.6: OpenSSL DTLS partial plaintext recovery: percentile-based success probabilities for recovering $P^*[15]$ assuming $P^*[14]$ is known, $n = 10$.

We also conducted a 2-byte recovery attack against OpenSSL DTLS; this attack is effectively the first step of the full plaintext recovery attack described in Section 5.4. Figure 5.7 shows the success probabilities for recovering $P^*[14]$ and $P^*[15]$ when $n = 10$. Again, the attack is very effective, recovering both bytes with success probability 0.93 for 2^{19} trials ($L = 2^3$). The quality of these results is evidence that the attack should extend easily to a full plaintext recovery attack. Figure 5.8 shows our results for $n = 1$, which we recall serves as an experimental model for TLS. We see that 2-byte recovery is reliable given 2^{23} trials ($L = 2^7$); we already reach more than 80% success rate using 2^{22} trials.

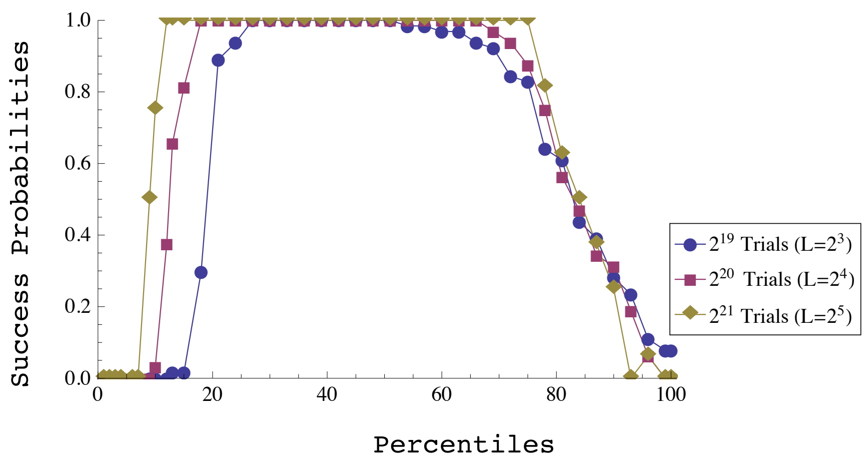


Figure 5.7: OpenSSL DTLS 2-byte recovery: percentile-based success probabilities for recovering $P^*[14]$ and $P^*[15]$, $n = 10$.

5.5.6 More Challenging Network Environments

We have not conducted experiments where the attacker is not situated in the same LAN as the server. Given the small timing differences involved, we would expect the attacks to fail, in the case of TLS, when the attacker is remote, i.e. more than a couple of hops away from the server, or that very large numbers of sessions would be needed to get reliable results. Nevertheless, there are realistic scenarios where the proximity requirement can be met, for example when a hostile network service provider attacks its customers, or in cloud computing environments. For DTLS, the timing signals can be amplified, effectively by an arbitrary amount, and so we would expect to be able to mount the attacks remotely.

5.6 Other Implementations of TLS

5.6.1 GnuTLS

The GnuTLS implementation of MEE-TLS-CBC deals with bad padding in a different way to that recommended in the RFCs: instead of assuming zero-length padding, it uses the last byte of plaintext to determine how many plaintext bytes to remove (whether or not those bytes are correctly formatted padding). More precisely, GnuTLS sets a variable `pad` as:

```
pad = ciphertext->data[ciphertext->size - 1] + 1
```

and then, after doing some basic sanity checking on the value of `pad`, subtracts `pad` bytes from the length field:

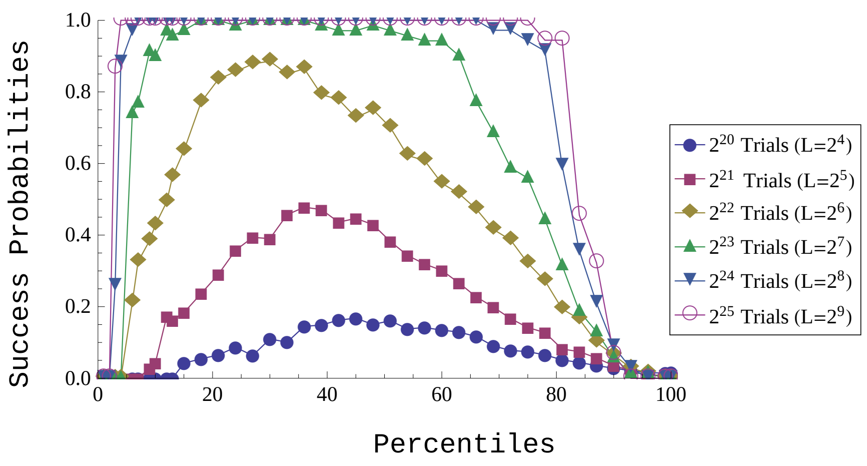


Figure 5.8: OpenSSL DTLS 2-byte recovery: percentile-based success probabilities for recovering $P^*[14]$ and $P^*[15]$, $n = 1$.

```
length = ciphertext->size - tag_size - pad
```

The GnuTLS code then proceeds to check the padding bytes, but the value of `length` stays the same for the remainder of the processing whether the padding check succeeds or fails. This variable dictates the number of record bytes involved in the MAC verification.

Since this approach is a natural alternative to the RFCs' advice for handling bad padding, we analyse it in detail, first for HMAC-SHA-1 as the MAC algorithm, and then in brief for other MAC algorithms. As before, we assume that our block cipher is AES and that IVs are explicit, with obvious modifications for other cases. We focus on TLS, but our attacks apply equally to DTLS. We then report experimental results.

GnuTLS + HMAC-SHA-1: Firstly, we point out that GnuTLS-style processing is just as vulnerable to distinguishing attacks as RFC-compliant processing. Indeed, the attack described in Section 5.3 will work just as before[4]. We next present an attack that recovers the rightmost byte of plaintext in any target block for GnuTLS-style padding processing.

Let C^* denoting the target ciphertext block, C' denote the previous ciphertext block and Δ denote a mask block of 16 bytes. We consider the decryption of a ciphertext $C^{\mathrm{att}}(\Delta)$ of the form:

$$C^{\mathrm{att}}(\Delta) = \mathtt{HDR} \,||\, C_0 \,||\, C_1 \,||\, C_2 \,||\, \ldots \,||\, C_{18} \,||\, C' \oplus \Delta \,||\, C^*$$

[4]In fact, since the attack only involves plaintexts which are correctly padded, it will work for *any* correct decryption algorithm, that does not implement special countermeasures.

in which there are 20 non-IV ciphertext blocks, the penultimate block is an XOR-masked version of C' and the last block is C^*, the target ciphertext block. The corresponding 320-byte plaintext is $P = P_1 \mid\mid P_2 \mid\mid \ldots \mid\mid P_{19} \mid\mid P_{20}$ in which

$$\begin{aligned} P_{20} &= D_{K_e}(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Now we need consider only two distinct cases, which between them cover all possibilities:

1. P_{20} ends with a `0x00` byte: in this case, a single byte of padding is removed, the next 20 bytes are interpreted as a MAC tag T, and the remaining $320 - 21 = 299$ bytes of plaintext are taken as the record R. MAC verification is then performed on a $13 + 299 = 312$-byte message `SQN` $\mid\mid$ `HDR` $\mid\mid R$.

2. P_{20} ends with any other byte value: in this case, at least two bytes of "padding" are removed, the next 20 bytes are interpreted as a MAC tag T, and the remaining bytes of plaintext are taken as the record R. Because the starting message length, at 320 bytes, is long enough to allow for the removal of 256 bytes of padding and a 20-byte MAC whilst still leaving a non-null record, no length sanity tests will fail. MAC verification is then performed on a message `SQN` $\mid\mid$ `HDR` $\mid\mid R$ that contains *at most* 311 bytes.

In both cases, the MAC verification will fail (with overwhelming probability) and an error message produced. Notice that, in accordance with the discussion in Section 5.2, in Case 1, the MAC verification will involve 9 evaluations of the compression function for SHA-1, while Case 2 requires at most 8 evaluations. Therefore, we can hope to distinguish the two cases by careful timing, as previously.

Now the single-byte plaintext recovery attack is straightforward: the attacker injects a sequence of ciphertexts $C^{\text{att}}(\Delta)$ with values of Δ that vary over all possible values in the last byte $\Delta[15]$, then (in the worst case) after 2^8 trials, the attacker will surely select a value for Δ such that $C^{\text{att}}(\Delta)$ triggers Case 1. When this is detected, he knows that P_{20} ends with a `0x00` byte and can infer the value of the last byte of P^* via the blockwise equation $P_{20} = P^* \oplus \Delta$.

This basic attack can be further improved. The 2 most significant bits of the last byte of P^* can be extracted using 4 trials by simply examining the time taken to produce an error message when ciphertexts $C^{\text{att}}(\Delta)$ are injected for values Δ which vary in the 2 most significant bits of $\Delta[15]$: the maximum running time is produced when the last byte of P_{20} is set to have bits 00 in the most significant positions. The remaining 6 bits can then be extracted using a further 64 trials to find the value of $\Delta[15]$ which triggers Case 1. Thus an enhanced version of the attack only needs 68 trials to recover the last byte of the target block.

For TLS, the usual problems of fatal errors and noisy timing information can be overcome in a multi-session attack. For DTLS, we can use the techniques from Chapter 4 to amplify the timing differences and overcome the lack of error messages.

GnuTLS + HMAC-MD5/HMAC-SHA-256: For HMAC-MD5 and HMAC-SHA-256, a similar analysis as before shows that the ciphertext $C^{\text{att}}(\Delta)$ triggers "slow" MAC evaluation (9 compression function evaluations) if P_{20} has last byte that is any of the 5 possibilities $\texttt{0x00}, \texttt{0x01}, \texttt{0x02}, \texttt{0x03}, \texttt{0x04}$, while all other values for the last byte of P_{20} result in "fast" MAC evaluation (at most 8 evaluations). These 5 byte values correspond to bit patterns $000, 001, 010, 011, 100$ in the 3 least significant bits. Exploiting this, we can build an attack using even fewer trials than previously. For TLS, we will need a multi-session attack, but note that the parameter L can be quite small since we only need to distinguish between a few possibilities (at most 16) in each phase of the attack. We omit the details.

Interestingly, the attacks for HMAC-MD5 and HMAC-SHA-256 are much more efficient for GnuTLS-style processing than they are for RFC-compliant processing. This is opposite to the situation for HMAC-SHA-1. We note that we have not found attacks for GnuTLS-style processing that can extract more than the last byte of the target block. This is not surprising in view of the fact that the decryption time for GnuTLS-style processing depends only on the last byte of plaintext.

Attack implementation for GnuTLS: We worked with version 3.0.21 of GnuTLS to implement the above attacks. In doing so, we found some subtle coding errors.

Firstly, the variable `pad` is defined as being of type `uint8`. In the code:

```
pad = ciphertext->data[ciphertext->size - 1] + 1
```

this has the unintended action of setting `pad` to zero when the last byte of plaintext equals `0xFF` instead of the desired value of 256, meaning that no bytes of padding are removed in this case instead of 256 bytes. As a consequence, GnuTLS does not properly support variable length padding during decryption, and the TLS session would be terminated if the encrypting party ever uses `0xFF` padding.

This coding error is easily patched, but means that our attacks do not quite work as described, since now 2 byte values (`0x00` and `0xFF`) in the last byte of P_{20} lead to slow MAC verification (in the HMAC-SHA-1 case). In fact, this does not present a serious barrier to our attack, and there is a variant using at most 66 trials to recover the last byte of P^*.

The second coding error we found relates to the implementation of the padding check. This uses the following `for` loop:

```
for (i = 2; i < pad; i++)
    {
      if (ciphertext->data[ciphertext->size - i] !=
          ciphertext->data[ciphertext->size - 1])
        pad_failed = GNUTLS_E_DECRYPTION_FAILED;
    }
```

It is not hard to see that this loop should also cover the edge case `i=pad` in order to carry out a full padding check. This means that one byte of what should be padding actually has a free format. This would enable, for example, a variant of the short MAC attack of [74] even if variable length padding was not supported. This coding error does not affect our attack. Notice also that the number of iterations in the loop depends on `pad`, which is plaintext-dependent.

Experimental Results for GnuTLS: By default, GnuTLS adds random length padding to every TLS record it sends (including alerts), subject to constraints imposed by the TLS specification. The time required to encrypt that random padding disrupts the timing signal that our attacks attempt to detect. For the purposes of experimental validation, we disabled GnuTLS's random padding. Note, however, that the attacks would still be effective even if the random padding were to be reactivated, since the error messages can be grouped according to their lengths, and the time difference attributable to adding extra padding can be profiled and subtracted for each group.

We began by measuring the time (in hardware cycles) taken by the GnuTLS server to perform the padding check, MAC verification and other associated operations as a function of the value of $\Delta[15]$, for ciphertexts containing 20 non-IV blocks and with the last byte of P^* equal to 0x00. Figure 5.9 shows the results. The expected behaviour is observed: byte values 0x00 and 0xFF have similar, long processing times. Moreover, there are four "blocks" of timings, corresponding to the reducing number of compression function evaluations needed as the byte value $\Delta[15] \oplus P^*[15]$ increases. (Here, $P^*[15]$ denotes the last byte of the target plaintext block P^*.) Within these blocks, the trend is upwards, and this is attributable to the increasing amount of time needed for the padding check as the value of `pad` increases.

Our next step was to gather timing of error messages from the network. Figure 5.10 shows median network timings for the same ciphertext structure. It is evident

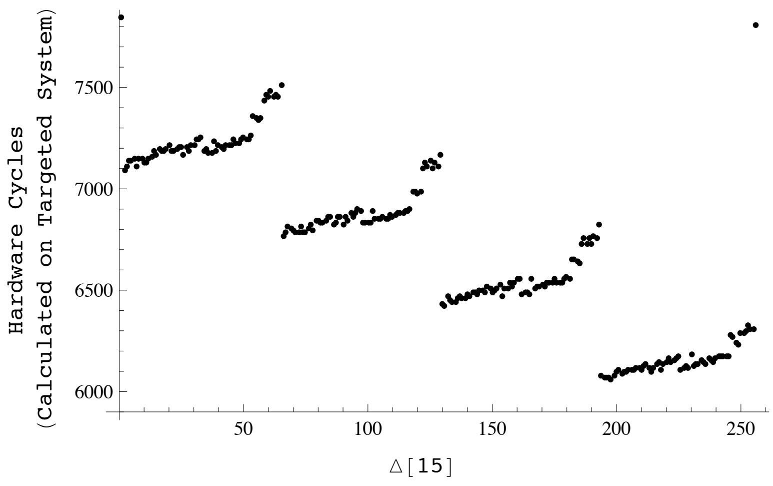


Figure 5.9: GnuTLS TLS median server timings (in hardware cycles) for varying values of $\Delta[15]$ and $P^*[15] =$ 0x00.

that there are anomalies at byte values 0x01, 0x11, ..., 0xF1 (with 16 byte increments). In further testing, we discovered that their positions did not depend on the plaintext byte $P^*[15]$. This phenomenon was subsequently explained to us [61] as arising from the way in which GnuTLS's random number generator updates its state (when generating CBC-mode IVs for TLS's encrypted error messages). We handled this in our attack by setting the timing values for these mask values to the average value of the neighbouring bytes.

The data is clearly very noisy, and the distinct pattern exhibited in the server timings in Figure 5.9 is not immediately evident in Figure 5.10. However, a zoomed view (see Figure 5.11) shows that an overall descending pattern is evident. Further analysis using linear regression shows that the ascending pattern within each of the 4 blocks is weakly preserved in the network timings (see Figure 5.12). We could not reliably distinguish the values 0x00 and 0xFF needed for the attack mentioned above; however, we are able to reliably extract the 4 most significant bits (MSBs) of $P^*[15]$, as we explain briefly next. It is worth noting that after sharing our findings, the GnuTLS development team conducted similar experiments and reached the same conclusions as us, confirming our findings; they published their results in [60].

Extracting 4 bits of $P^*[15]$: To extract the 2 MSBs of $P^*[15]$, the attacker focusses on the overall downward trend in the processing time (as a function of $\Delta[15] \oplus P^*[15]$) exhibited in Figure 5.11. Let $\delta_7\delta_6 \ldots \delta_0$ denote the bits of $\Delta[15]$. By setting $\delta_7 = 0$ and then $\delta_7 = 1$, the attacker has 2 sets each containing 128 masks; he gathers timings for each of these two sets; if larger timings are obtained on average when $\delta_7 = 0$, then the

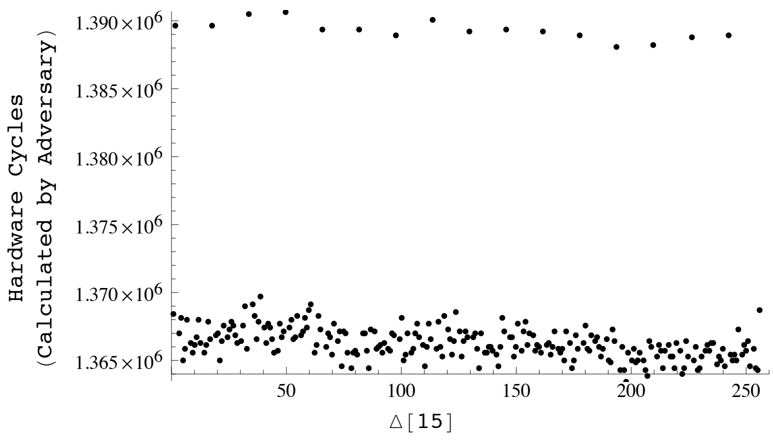


Figure 5.10: GnuTLS TLS median network timings (in hardware cycles) for varying values of $\Delta[15]$ and $P^*[15] = \mathsf{0x00}$.

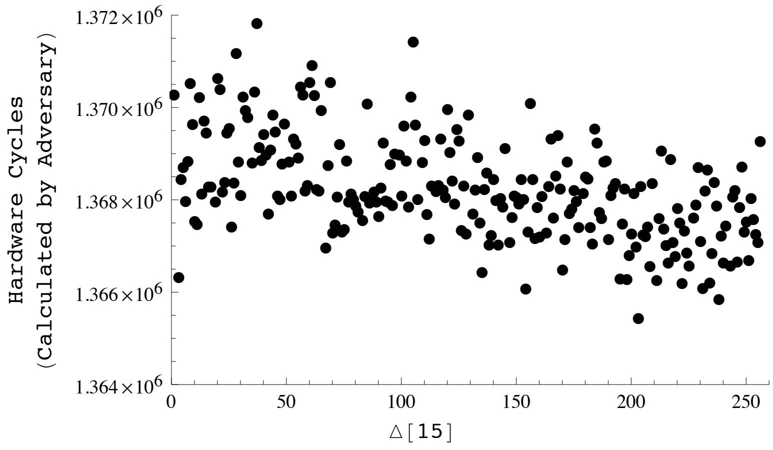
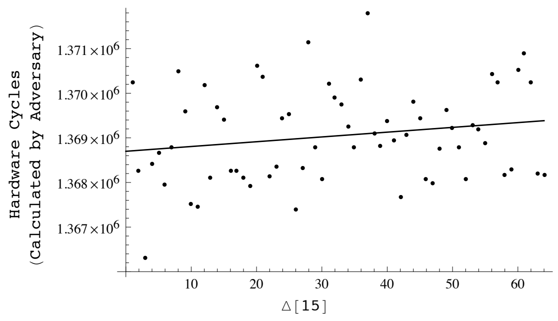
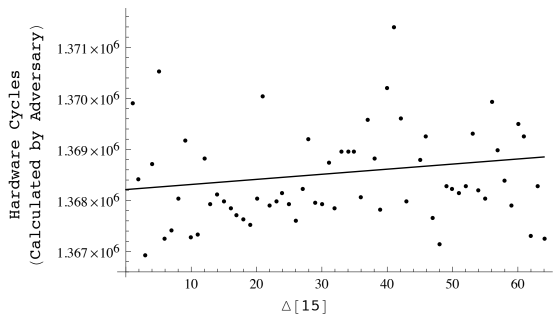


Figure 5.11: Zoomed view of GnuTLS TLS network timings.

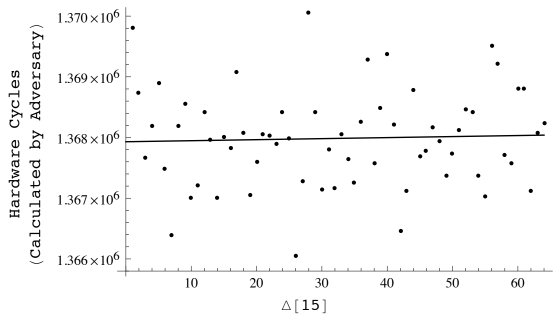
attacker deduces that the MSB of $P^*[15]$ is a 0; otherwise he guesses that the MSB is 1. The attacker can also use a reduced set of masks, and collect multiple timing samples for each mask that he tries. Thus we have two parameters: the total number of masks S that he uses across the two sets, and the number of timing samples L for each mask. The second MSB of $P^*[15]$ is extracted in the same way: now we consider masks for $\delta_6 = 0$ and then $\delta_6 = 1$. In principle, we have S as large as 256 again, by varying δ_7 as well as the other 6 bits of $\Delta[15]$. In practice, we just set $\delta_7 = 0$ when extracting the second MSB. The third and fourth MSBs are extracted in roughly the same way, but now we reverse the test, setting the targeted bit to 1 if larger timings are obtained on average when $\delta_5 = 0$ or $\delta_4 = 0$, respectively. This change reflects the ascending trend within the 4 blocks observed in Figure 5.9.



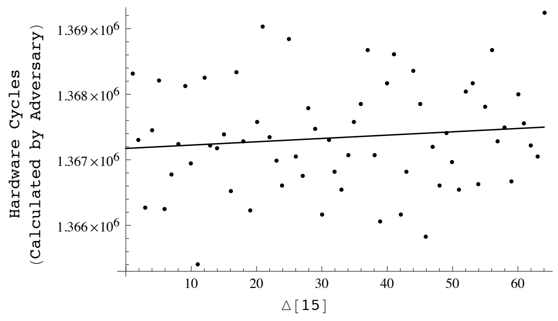
(a) 1st block



(b) 2nd block



(c) 3rd block



(d) 4th block

Figure 5.12: Zoomed view of GnuTLS TLS network timings for each of the 4 blocks.

Success probabilities for this attack are shown in Table 5.2. We tried to recover the remaining bits, but did not obtain significant success probabilities. Whilst extracting less plaintext than our OpenSSL attack, far fewer TLS sessions are required in this attack on GnuTLS. This indicates that ignoring the recommendations of the RFCs can have severe security consequences.

S \ L	**4**	**8**	**16**	**32**	**64**	**128**
4	0.575	0.662	0.746	0.828	0.875	0.937
8	0.516	0.615	0.781	0.836	0.844	1
16	0.531	0.609	0.766	0.852	0.969	1
32	0.536	0.596	0.750	0.898	0.984	1
64	0.544	0.596	0.781	0.937	0.984	1
128	0.555	0.627	0.812	0.977	1	1
256	0.593	0.635	0.859	1	1	1

MSB

S \ L	**4**	**8**	**16**	**32**	**64**	**128**
4	0.511	0.580	0.629	0.687	0.656	0.812
8	0.513	0.576	0.695	0.789	0.812	0.812
16	0.515	0.564	0.637	0.742	0.734	0.844
32	0.509	0.549	0.617	0.734	0.766	0.844
64	0.519	0.570	0.656	0.859	0.953	0.969
128	0.544	0.557	0.557	0.914	1	1

Second MSB

S \ L	**4**	**8**	**16**	**32**	**64**	**128**
4	0.486	0.451	0.418	0.391	0.422	0.375
8	0.522	0.508	0.523	0.500	0.531	0.625
16	0.537	0.555	0.598	0.625	0.625	0.781
32	0.543	0.572	0.609	0.609	0.609	0.609
64	0.528	0.541	0.602	0.758	0.758	1

Third MSB

S \ L	**4**	**8**	**16**	**32**	**64**	**128**
4	0.456	0.434	0.363	0.336	0.312	0.25
8	0.487	0.484	0.445	0.477	0.484	0.375
16	0.495	0.531	0.539	0.570	0.594	0.687
32	0.506	0.520	0.566	0.695	0.828	0.812

Fourth MSB

Table 5.2: GnuTLS success probabilities for recovering the four MSBs of $P^*[15]$.

5.6.2 Further Implementations

NSS: Network Security Services (NSS)[5] is an open-source set of libraries implementing, amongst other things, TLS. It is widely used, including in Mozilla client products and Google Chrome.

In the decryption code[6] the variable `plaintext->len` is reduced by the assumed amount of padding (`padding_length + 1`) *before* the padding is checked for correctness. This is the same approach as taken in GnuTLS, potentially rendering the code vulnerable to an attack recovering a single byte of plaintext per block. The sanity check performed at the beginning of the decryption code is also problematic, since it leaves `plaintext->len` unmodified if the check fails, meaning that MAC verification may take longer than when the check passes.

PolarSSL: We also examined the PolarSSL[7] implementation of TLS. The code[8] behaves in much the same way as OpenSSL, setting a variable `padlen` to 0 if the padding check fails, and then verifying the MAC on a record stripped of `padlen` bytes. This would render it vulnerable to the attacks described in Section 5.4.

In fact, this implementation has other problems too. The code does not sanity check `padlen` before running the padding check, meaning that out-of-bounds comparisons may be made if the value of `padlen` exceeds the plaintext length. It does sanity check `padlen` *after* the padding check, checking that the plaintext is big enough to contain both the expected amount of padding and the MAC tag. However, it does not perform any MAC check if this sanity check fails, but instead exits immediately. This would render the implementation vulnerable to a simple timing-based distinguishing attack as follows: M_0 consists of 256 copies of 0xFF, while M_1 consists of 255 arbitrary bytes followed by 0x00; as in the attack of Section 5.3, the encrypted version C of one of these is received; the attacker truncates C so that the underlying plaintext has 256 bytes; if the message was M_0, then the padding is good, but the post-padding sanity check fails and no MAC computation is performed; if the message was M_1, then the padding is also good, but now the post-padding sanity check passes and a MAC computation is performed. This attack produces a larger timing difference than our previous distinguishing attack and illustrates the role that careful sanity checking plays in preventing attacks.

[5] `http://www.mozilla.org/projects/security/pki/nss`

[6] We worked with version 3.13.6 available at `https://ftp.mozilla.org/pub/mozilla.org/security/nss/releases/NSS_3_13_6_RTM/src/`.

[7] `polarssl.org/`

[8] We worked with version 1.1.4 available at `http://polarssl.org/trac/browser/trunk/library/ssl_tls.c`.

However, none of these attacks would work in practice, since in its default configuration, PolarSSL does not send any TLS alert messages when decryption errors are encountered. This means that PolarSSL is *not* RFC-compliant in this aspect, since such alerts are a required part of TLS implementations.

yaSSL: The yaSSL[9] embedded SSL library, CyaSSL, is targeted at embedded and real-time operating system environments. It appears to have rather few known vulnerabilities, with only 5 being reported in the CVE database[10] since 2005. The CyaSSL code[11] does not perform proper padding checks, but instead just examines the last byte of plaintext and uses this to determine how many bytes to remove. This can be seen in the following CyaSSL code extract:

```
if (ssl->specs.cipher_type == block) {
 if (ssl->options.tls1_1)
    ivExtra = ssl->specs.block_size;
 pad = *(input + idx + msgSz - ivExtra - 1);
 padByte = 1;
}

dataSz = msgSz - ivExtra - digestSz - pad - padByte;
if (dataSz < 0) {
 CYASSL_MSG("App data buffer error, malicious input?");
 return BUFFER_ERROR;
}
```

This approach renders the code vulnerable to the old attack from [68] which recovers one byte of plaintext per block. This was the only implementation that we found that still contains this basic flaw. Note also that the sanity checking represented by the last 3 lines of code above would render the code vulnerable to other plaintext recovery attacks even if the padding check was done properly, since it exits the code without performing a MAC check if the tested condition (which depends on the byte `pad` extracted from the plaintext) is violated.

[9]http://yassl.com/yaSSL/Home.html

[10]http://www.cvedetails.com/vulnerability-list/vendor_id-3485/Yassl.html

[11]We worked with version 2.3.0 available at http://yassl.com/yaSSL/Source/output/src/internal.c.html.

Java: We have examined the BouncyCastle[12] and OpenJDK[13] Java implementations of TLS.

The BouncyCastle code does careful sanity checking of the padding length (as indicated by the last byte of plaintext) but treats the padding as having length 1 if the padding format, when checked, is found to be incorrect (a variable `paddingsize` is set to 0, but then the plaintext size is reduced by an amount `paddingsize+minLength` where `minLength` is set to be 1 larger than the MAC tag size). This deviates slightly from the recommendation of the RFCs to treat the padding as having length zero, but still allows our attacks in Sections 5.3 and 5.4 to be applied (for Case 3 of the main plaintext recovery attack in Section 5.4, MAC verification ends up being performed on a 56-byte message, but this will still involve 5 evaluations of the compression function for SHA-1).

The OpenJDK code appears follow the recommendation of the RFCs in treating the padding as having zero length if the padding format, when checked, is found to be incorrect. This is because this case is trapped by exception handling, during which the variable defining the plaintext length is not changed. This potentially renders it vulnerable to our attacks in Sections 5.3 and 5.4.

Other implementations: There are further open-source and many closed-source implementations of (D)TLS. We have not conducted any further testing to see if these are vulnerable to any of our attacks. However, we expect that any RFC-compliant implementation will be vulnerable. We also expect that all implementations will be vulnerable to simple variants of our attacks, unless the implementers have taken great care to ensure that the decryption processing time is uniform, or nearly so. Our experiences in investigating open-source implementations suggests this is unlikely.

5.7 Countermeasures

Add Random Time Delays: A natural reaction to timing-based attacks is to add random time delays to the decryption process to frustrate statistical analysis. In fact, this countermeasure is surprisingly *in*effective, as we explain next.

Consider our distinguishing attack: this attack involves distinguishing two distributions X, Y, where X has mean μ and Y has mean $\mu + 4$, where we measure time

[12]`http://www.bouncycastle.org/viewcvs/viewcvs.cgi/java/crypto/src/org/bouncycastle/crypto/tls/TlsBlockCipher.java?view=markup`

[13]`http://hg.openjdk.java.net/jdk7/l10n/jdk/file/3598d6eb087c/src/share/classes/sun/security/ssl/SSLSocketImpl.java` and `http://hg.openjdk.java.net/jdk7/2d/jdk/file/85fe3cd9d6f9/src/share/classes/sun/security/ssl/CipherBox.java`

in units of compression function evaluations. Suppose X, Y both have variance σ^2. Now suppose we add a random delay that is uniformly chosen from the interval $[0, T]$ to the decryption process. Then we obtain distributions X', Y' with means $\mu + T/2$ and $\mu + 4 + T/2$ and variance $\sigma^2 + (T^2 - 1)/12$. Now consider the random variables $V_L = \sum_{i=1}^{L} X'_i/L$ and $W_L = \sum_{i=1}^{L} Y'_i/L$ obtained from averaging L samples of X', Y', respectively. Treating these samples as being independent, the Central Limit Theorem guarantees that V_L, W_L are approximately Normal with means $\mu + T/2$, $\mu + 4 + T/2$ and equal variance $\tau^2 = (\sigma^2 + (T^2 - 1)/12)/L$. Note that the difference between the means of V_L, W_L is 4; now, using standard results about the Normal distribution, it is easy to see that if $4\tau \leq 4$, then the distributions of V_L, W_L are sufficiently "tight" about their means that a simple statistical test based on taking means of L samples will be 90% accurate. Solving for L, we see that we need

$$L \geq \sigma^2 + (T^2 - 1)/12$$

and it is apparent that the effect of adding the random time delay is to increase the number of samples needed from σ^2 to $\sigma^2 + (T^2 - 1)/12$. From our experiments for OpenSSL, we estimate that $\sigma \approx 10$; then taking $T = 50$ only increases the number of samples needed for a 90% success rate from 100 to about 300, at the cost of increasing the average decryption time by 25 compression function evaluations. This does not seem like a good trade-off between security and performance.

Use RC4: The simplest countermeasure for TLS is to switch to using the RC4 stream cipher in place of CBC-mode encryption. However, this is not an option for DTLS. When a stream cipher is used in TLS, no padding is required. Consequently none of the attacks in this chapter will work. RC4 is widely supported in implementations of TLS, the same countermeasure is effective against the BEAST attack, and was fairly widely adopted in response to BEAST (e.g. by Google and Facebook). The use of a stream cipher in a MEE construction is well-supported by theory [57]. There are two potential drawbacks of making this switch. Firstly, the use of variable length padding in CBC-mode allows for a modicum of plaintext length hiding, and this is no longer possible when using a stream cipher. Secondly, and more importantly, the first bytes of keystream output by the RC4 generator have certain small biases, and TLS does not seem to discard these before starting encryption [94]. Recently, the authors of [4] presented new biases in the RC4 keystream output, as described in Chapter 3, making RC4 less likely to be a feasible long-term replacement of CBC-mode encryption for TLS.

Use Authenticated Encryption: Another possibility is to switch from MEE-TLS-CBC to using a dedicated authenticated encryption algorithm, such as AES-GCM or AES-CCM which were standardised for use in TLS in RFCs 5288 [90] and 6655 [63], respectively. In theory, this should obviate all attacks based on weaknesses in the MEE construction. However, we cannot rule out implementation errors, and we are not aware of any detailed analysis of implementations of these algorithms in (D)TLS for potential side-channels. A further issue is that authenticated encryption was only added in TLS 1.2, and this version of TLS is not yet widely supported in implementations. Finally, the current authenticated encryption algorithms do not offer any length-hiding facility.

Careful implementation of MEE-TLS-CBC decryption: Our final option is to encourage more careful implementation of MEE-TLS-CBC decryption. However, we believe that implementers will find it difficult to do this in a way that eliminates all significant timing channels (especially for DTLS).

The key requirement is to ensure constant processing time for all MEE-TLS-CBC ciphertexts of a given size. That is, the total processing time should depend only on the ciphertext size, and not on any characteristics of the underlying plaintext (including padding). The basic principle to be followed in achieving this is quite simple: since the major timing differences arise from MAC processing, implementations should make sure the same amount of MAC processing is carried out no matter what the underlying plaintext indicates the message length to be.

However, this simple principle is complicated by the need to also perform careful sanity checking on the underlying plaintext whilst avoiding the introduction of yet more timing side-channels, and to make sure appropriate amounts of MAC processing are performed even when these checks fail.

A further complication arises because the number of bytes to be examined in the padding check depends on the last byte of the last plaintext block, and so, even if the MAC processing is made uniform, the running time of the padding check may still leak a small amount of information about the plaintext. This can be seen for GnuTLS in Figure 5.9: notice that the maximum difference in the running time for the padding check is more than 1000 hardware cycles for this implementation. For example, then, distinguishing attacks would require a timing resolution of around 1000 hardware cycles, while a timing resolution of 250 cycles would be sufficient to allow an attack recovering 2 bits of plaintext per block for this implementation.

With these remarks in mind, we now proceed to give a detailed prescription of how to achieve constant-time processing of MEE-TLS-CBC ciphertexts, incorporating suitable sanity checking. In what follows, we let `plen` denote the length (in bytes) of the plaintext P obtained immediately after CBC-mode decryption of the ciphertext, `padlen` denote the last byte of that plaintext interpreted as an integer between 0 and 255, and t denote the length of the MAC tags (in bytes). Also, let `HDR`, `SQN` denote the (D)TLS record header and the expected value of the sequence number for this record. Our recommended procedure is then as follows:

1. First sanity check the ciphertext: check that its length in bytes is a multiple of the block-size b and is at least $\max\{b, t+1\}$ (for chained IVs) or $b+\max\{b, t+1\}$ (for explicit IVs). If these conditions are not met, then return `fatal error`.
2. Decrypt the ciphertext to obtain plaintext P; now `plen` will be a multiple of b and at least $\max\{b, t+1\}$.
3. If $t + \texttt{padlen} + 1 > \texttt{plen}$, then the plaintext is not long enough to contain the padding (as indicated by the last byte of plaintext) plus a MAC tag. In this case, run a loop as if there were 256 bytes of padding, with a dummy check in each iteration. Then let P' denote the first $\texttt{plen} - t$ bytes of P, compute a MAC on $\texttt{SQN} \,||\, \texttt{HDR} \,||\, P'$ and do a constant-time comparison of the computed MAC with the last t bytes of P. Return `fatal error`.
4. Otherwise (when $t+\texttt{padlen}+1 \leq \texttt{plen}$), check the last $\texttt{padlen}+1$ bytes of P to ensure they are all equal (to the last byte of P), ensuring that the loop does check all the bytes (and does not stop as soon as the first mismatch is detected). If this fails, then run a loop as if there were $256 - \texttt{padlen} - 1$ bytes of padding, with a dummy check in each iteration, and then do a MAC check as in the previous step. Return `fatal error`.
5. Otherwise (the padding is now correctly formatted) run a loop as if there were $256-\texttt{padlen}-1$ bytes of padding, doing a dummy check in each iteration. Then let P' denote the first $\texttt{plen}-\texttt{padlen}-1-t$ bytes of P, and let T denote the next t bytes of P (the remainder of P is valid padding). Run the MAC computation on $\texttt{SQN} \,||\, \texttt{HDR} \,||\, P'$ to obtain a MAC tag T'. Then set $L_1 = 13 + \texttt{plen} - t$, $L_2 = 13 + \texttt{plen} - \texttt{padlen} - 1 - t$, and perform an additional $\lceil \frac{L_1-55}{64} \rceil - \lceil \frac{L_2-55}{64} \rceil$ MAC compression function evaluations (on dummy data). Finally, do a constant-time comparison of T and T'. If these are equal, then return P'. Otherwise, return `fatal error`.

When implementing the above procedure, it would be tempting to omit seemingly unnecessary computations that are performed, for example when $t + \texttt{padlen} + 1 > \texttt{plen}$. However, these are needed to prevent other timing side-channels like those reported in Chapter 4 for the GnuTLS implementation of DTLS. Notice also that the

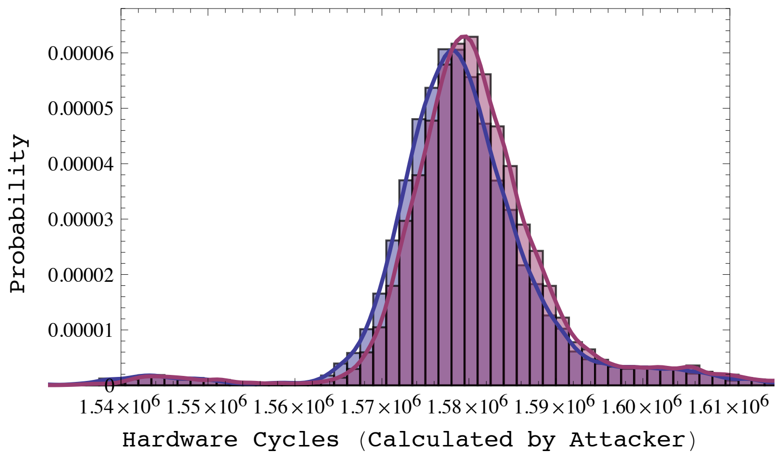


Figure 5.13: Distribution of timing values (outliers removed) for distinguishing attack on OpenSSL TLS, using our decryption procedure.

dummy computations performed in the last step are compression function evaluations and not full MAC computations. These give a MAC computation time that is the same irrespective of how much padding is removed (and equal to that carried out in earlier steps). Finally, note that some adjustments to this procedure would be needed when SHA-384 is used as the hash function in HMAC: SHA-384 operates on 128-byte blocks and uses a 16-byte encoding for message length.

We have implemented the above procedure by modifying OpenSSL version 1.0.1, the same version used for our attacks. We modified the code in files `ssl/s3_pkt.c` and `ssl/t1_enc.c` to perform the required sanity checks, dummy padding checks, and dummy MAC compression function evaluations. In `ssl/s3_pkt.c`, we make a single call to OpenSSL's `SHA1_Update` function using a message size that will invoke the required number of dummy MAC compression function evaluations. Our call to `SHA1_Update` happens before OpenSSL's actual MAC calculation and comparison operations.

We then ran our distinguishing attack from Section 5.3 against the modified code of OpenSSL. Each packet in the attack passes the padding check, but fails MAC verification, causing the server to close the TLS session and send an encrypted alert message. Figure 5.13 shows the distribution of timing values (in hardware cycles) after implementing our procedure. This figure should be compared with Figure 5.2: visual inspection alone shows that the timing difference is substantially reduced. In fact, the separation between the medians of the two distributions is reduced from about 8500 to about 1100 hardware cycles (from around 2.5μs to 0.32μs). In turn, this small separation means that 128 sessions are needed to achieve a distinguishing success probability of 0.68, whereas, prior to our modifications, just 1 session was enough to give a success probability of 0.756. For the plaintext recovery attack, the adversary will have access to timing differences roughly one quarter of this, i.e. roughly 80ns on our hardware. Notice also that the two distributions are reversed compared to Figure 5.2, i.e. processing `0xFF` packets now takes longer, on average, than for `0x00` packets. We believe that this is caused by overhead introduced by the `SHA1_Update` function call that occurs for `0xFF` packets but not `0x00` packets.

To achieve further reductions in timing difference would require a more sophisticated "constant time" programming approach. The OpenSSL patch addressing the attacks in this chapter provides an exemplar of how to do this. The complexity of the OpenSSL patch is notable, with around 500 lines of new 'C' code being required[14].

5.7.1 Disclosure

Given the large number of affected implementations, we first notified the IETF TLS Working Group chairs, the IETF Security Area directors and the IRTF Crypto Forum Research Group (CFRG) chairs of our attacks in November 2012. We then began the process of contacting individual vendors:

OpenSSL addressed the attacks in versions 1.0.1d, 1.0.0k and 0.9.8y[15].

NSS addressed the attacks in version 3.14.3[16].

Microsoft performed an investigation and determined that the issue had been adequately addressed in previous modifications to their TLS and DTLS implementations

Apple were notified of our attacks in December 2012. The status of patch development by Apple was not communicated to us.

GnuTLS corrected the programming errors in decryption that we identified in version 3.1.6 (released 02/01/2013) and addressed the attacks in versions 2.12.23, 3.0.28 and 3.1.7[17].

PolarSSL addressed the attacks in version 1.2.5[18].

CyaSSL addressed the attacks in CyaSSL version 2.5.0[19].

MatrixSSL addressed the attacks in version 3.4.1[20].

Opera addressed the attacks in Opera version 12.13[21].

[14] http://www.imperialviolet.org/2013/02/04/luckythirteen.html
[15] http://www.openssl.org/news/secadv_20130205.txt
[16] https://developer.mozilla.org/en-US/docs/NSS/NSS_3.14.3_release_notes
[17] http://www.gnutls.org/news.html
[18] https://polarssl.org/tech-updates/releases/polarssl-1.2.5-released
[19] http://www.yassl.com/yaSSL/Docs-cyassl-changelog.html
[20] http://matrixssl.org/news.html
[21] http://www.opera.com/docs/changelogs/unified/1213/

F5 were notified of the attacks in December 2012. They have informed us that their TLS data plane traffic is not vulnerable due to cryptographic offload, but that local management ports and virtual editions were vulnerable and the attacks were addressed for different F5 platforms[22].

BouncyCastle addressed the attacks in version 1.48 of the Java library[23]. The C# version of BouncyCastle was fixed in CVS at a similar time, and will be included in release 1.8 at a later date.

Oracle (Java) addressed the attacks as part of a special critical patch update of JavaSE[24].

In addition, a number of other companies and organisations were given advance notice of the attacks prior to them being made public.

5.8 Chapter Conclusion

We have demonstrated a variety of attacks against implementations of (D)TLS. We reiterate that the attacks are ciphertext-only, and so can be carried out by the standard MITM attacker, *without* a chosen-plaintext capability. The attacks that are possible depend crucially on low-level implementation details, as well as factors such as the relationship between the MAC tag size t and the block size b. All implementations we examined were vulnerable to one or more attacks.

For TLS, we need a multi-session attack, with, in some cases, many sessions. This limits the practicality of the attacks, but note that they be further improved using standard techniques such as language models and sequential estimation. They can also be enhanced in a BEAST-style attack to enable efficient recovery of HTTP cookies. The timing differences we must detect are close to or below the levels of jitter one typically finds in real networks. In particular, our attacker needs to be positioned relatively close (in terms of network hops) to the machine being attacked. Still, the attacks should be considered as a realistic threat to TLS, and we have described a range of suitable countermeasures. The attacks are much more serious for DTLS, because of this protocol's tolerance of errors and because of the availability of timing amplification techniques from Chapter 4. Very careful implementation of the MEE-TLS-CBC decryption algorithm is needed to thwart these amplification techniques. In view of this, we highly recommend the use of a suitable authenticated encryption algorithm in preference to CBC-mode for DTLS.

[22]http://support.f5.com/kb/en-us/solutions/public/14000/100/sol14190.html
[23]http://www.bouncycastle.org/latest_releases.html
[24]http://www.oracle.com/technetwork/topics/security/javacpufeb2013update-1905892.html

More generally, our attacks illustrate the difficulty of implementing MEE securely. Similar issues were identified for MEE configurations of IPsec in [30]. We encourage protocol designers in general, and the IETF TLS working group in particular, to move away from using MEE. None of the attacks on TLS presented here would have been possible with an Encrypt-then-MAC approach, for example. A more realistic solution for TLS is to move as quickly as possible to TLS 1.2 and adopt its authenticated encryption algorithms.

Chapter 6

Conclusion

6.1 Themes

In this thesis, we demonstrated how weaknesses in the design and implementation of three secure network protocols can be exploited using new techniques. For example:

The DepenDNS protocol suffers from a number operational definicies and is vulnerabe to cache poisoning and denial of service attacks under some assumptions.

The OpenSSL implementation of DTLS did not include the padding oracle attack countermeasure, due, most likely, to the assumption that the lack of error messages makes DTLS *irrelevant* to padding oracle attacks. This is an *implementation* decision that we exploited to perform a full plaintext recovery attack against the OpenSSL implementation of DTLS, discussed in Chapter 4.

The authors of the DTLS standards [82, 83] opted to make anti-replay an optional security feature. This is a *design* decision that greatly assisted us to speed up the attack against DTLS, discussed in Chapter 4.

The padding oracle attack countermeasures introduced in TLS 1.1 and 1.2, DTLS 1.0 and 1.2, and in implementations of TLS 1.0 and SSL 3.0, did not fully implement constant time processing. The way the countermeasures were constructed is a *design* decision that we exploited to create new plaintext recovery attacks, discussed in Chapter 5.

A number of TLS and DTLS implementations had trivial coding errors in some security checks, were not compliant with the standards, or did not implement countermeasures against *known* attacks. For example, the GnuTLS implementation of (D)TLS did not handle the padding check correctly. PolarSSL, on the other hand, did not generate TLS Alert messages as per the standard, while yaSSL was vulnerable to old TLS attacks. More examples are discussed in Chapter 5, demonstrating the current state of implementing (D)TLS in some popular open source libraries.

MEE is *hard* to analyse and *hard* to implement securely: The MAC-Encode-Encrypt construction has proven to be hard to implement securely with long history of attacks and fixes. In addition, despite all the effort, achieving constant time processing is difficult. The 500-line patch to address our attacks in OpenSSL demonstrates the amount of efforts and changes required to try to achieve constant time processing, eliminating side channel leakages in general.

Practicality: Our work on (D)TLS demonstrates the possibility of practicality implementing attacks that might seem purely theoretical in nature. We argue that our attacks against TLS and DTLS are on the verge of being practical.

Attacks only get better: In Chapter 5, we demonstrated how to reduce the complexity of all our attacks against (D)TLS, lowering the number of sessions required to recover an unknown byte from roughly 2^{23} to 2^{13}, by exploring the possibility of combining the original attack with the use of a JavaScript and targeting base64 encoded cookies.

Impact: The TLS community (the IETF TLS working group, open source code developers, vendors and researchers) reacted positively to our work. The impact of our work extends from patching most of the TLS implementations to further speeding up the deployment of TLS 1.2 and the adoption of authenticated encryption. For example, NSS, which is used in Firefox and Google Chrome, released in July 2013 version 3.15.1[1] that brings TLS 1.2 support, but without AES-GCM that is planned for a future release. In fact, a patch that implements AES-GCM has already been submitted to NSS[2] and is available in the beta of version 3.15.2[3].

Bridging a gap: During the period of discussing and implementing the countermeasures to our attacks in Chapter 5, we were surprised by the *lack* of collaboration, other than some individual initiatives, between the IETF TLS working group, TLS open source code developers, vendors and academic researchers. In the case of (D)TLS, we advocate for collaboration between the IETF TLS working group and the rest of the

[1]`https://developer.mozilla.org/en-US/docs/NSS/NSS_3.15.1_release_notes`
[2]`https://bugzilla.mozilla.org/show_bug.cgi?id=880543`
[3]`https://hg.mozilla.org/integration/mozilla-inbound/rev/096d62676298`

TLS community (researchers and software developers); we argue that our work is one step towards achieving this.

6.2 Areas of Further Research

Enhancing our attacks against (D)TLS is one area that could be explored further. One idea is to implement the attacks in combination with a language model as described in Chapter 5. There is also the opportunity to examine the possibility of mounting the attacks in a multi-hop network setup, combined with more sophisticated statistical methods than the ones we used.

Almost all of the open source (D)TLS libraries have implemented countermeasures for the attacks discussed in Chapter 5. An interesting question is whether the deployment of these fixes has *effectively* eliminated the side channels exploited in our attacks. Further work can be carried out to test this. In addition, there is the potential of discovering other *unknown* sources of side channel information in the current deployments of (D)TLS, and which could be exploited to construct new attacks against the protocols.

Analysing the AES-GCM implementation in open source (D)TLS libraries is another area of research, especially that authenticated encryption, supported in TLS 1.2, is now generally considered the natural replacement for MEE and RC4 in (D)TLS.

We advocate for further research in understanding the interaction between secure network protocols and their upper and lower-layer protocols, and its impact on the overall security level of a system. We believe that this is an area where further research is needed.

Bibliography

[1] D. E. 3rd. Domain Name System Security Extensions. RFC 2535, Internet Engineering Task Force, Mar. 1999.

[2] D. E. 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, Internet Engineering Task Force, Jan. 2011.

[3] D. E. 3rd and A. Panitz. Reserved Top Level DNS Names. RFC 2606, Internet Engineering Task Force, June 1999.

[4] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. Schuldt. On the Security of RC4 in TLS and WPA. In *USENIX Security Symposium*, 2013.

[5] N. AlFardan and K. G. Paterson. Plaintext-Recovery Attacks Against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[6] N. J. AlFardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[7] AlFardan, Nadhem J and Paterson, Kenneth G. An analysis of DepenDNS. In *Information Security*, pages 31–38. Springer, 2011.

[8] Alfredo Pironti and Pierre-Yves Strub and Karthikeyan Bhargavan. Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures. Technical Report 8067, INRIA, September 2012.

[9] P. Almquist. Type of Service in the Internet Protocol Suite. RFC 1349, Internet Engineering Task Force, July 1992.

[10] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, Internet Engineering Task Force, Mar. 2005.

[11] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, Internet Engineering Task Force, Mar. 2005.

[12] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, Internet Engineering Task Force, Mar. 2005.

[13] D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833, Internet Engineering Task Force, Aug. 2004.

[14] G. V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

[15] G. V. Bard. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. In *SECRYPT*, pages 99–109, 2006.

[16] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO 2012*, pages 608–625. Springer, 2012.

[17] A. Barth. HTTP State Management Mechanism. RFC 6265, Internet Engineering Task Force, Apr. 2011.

[18] R. Bellis. DNS Transport over TCP – Implementation Requirements. RFC 5966, Internet Engineering Task Force, Aug. 2010.

[19] J. Benaloh, B. Lampson, D. Simon, T. Spies, and B. Yee. The Private Communication Technology (PCT) Protocol. *Microsoft Developer's Network CD, Internet Draft, Microsoft Corporation*, 1995.

[20] G. Bertoni, J. Daemen, M. Peeters, and G. Assche. The Keccak SHA-3 Submission. Submission to NIST (Round 3)(2011).

[21] A. Bhimani. Securing the Commercial Internet. *Communications of the ACM*, 39(6):29–35, 1996.

[22] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Internet Engineering Task Force, Oct. 1989.

[23] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5):701–716, 2005.

[24] P. Calhoun, M. Montemurro, and D. Stanley. Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification. RFC 5415, Internet Engineering Task Force, Mar. 2009.

[25] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2003.

[26] G. Chuanxiong and Z. Shaoren. Analysis and Evaluation of the TCP/IP Protocol Stack of LINUX. In *Communication Technology Proceedings, 2000. WCC-ICCT 2000. International Conference on*, volume 1, pages 444–453. IEEE, 2000.

[27] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, Internet Engineering Task Force, Aug. 2011.

[28] S. A. Crosby, D. S. Wallach, and R. H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[29] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS Forgery Resistance Through 0x20-Bit Encoding. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 211–222. ACM, 2008.

[30] J. P. Degabriele and K. G. Paterson. Attacking the IPsec Standards in Encryption-only Configurations. In *IEEE Symposium on Security and Privacy*, pages 335–349. IEEE Computer Society, 2007.

[31] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.

[32] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006.

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

[34] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614, Internet Engineering Task Force, Sept. 2006.

[35] T. Duong and J. Rizzo. Here come the $\oplus$ Ninjas. Unpublished manuscript, 2011.

[36] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM Conference on Computer and Communications Security*, 2012.

[37] S. R. Fluhrer and D. A. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In *FSE*, pages 19–30, 2000.

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

[39] S. Friedl. An Illustrated Guide to the Kaminsky DNS Vulnerability. http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html.

[40] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *ACM Conference on Computer and Communications Security*, 2012.

[41] Y. Gluck, N. Harris, and A. Prado. BREACH: Reviving The Crime Attack. http://breachattack.com.

[42] F. Gont and S. Bellovin. Defending against Sequence Number Attacks. RFC 6528, Internet Engineering Task Force, Feb. 2012.

[43] F. Gont and A. Yourtchenko. On the Implementation of the TCP Urgent Mechanism. RFC 6093, Internet Engineering Task Force, Jan. 2011.

[44] W. Hardaker. Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP). RFC 5953, Internet Engineering Task Force, Aug. 2010.

[45] P. Hoffman. Cryptographic Algorithm Identifier Allocation for DNSSEC. RFC 6014, Internet Engineering Task Force, Nov. 2010.

[46] A. Hubert and R. van Mook. Measures for Making DNS More Resilient Against Forged Answers. RFC 5452, Internet Engineering Task Force, Jan. 2009.

[47] T. Iwata and K. Kurosawa. OMAC: One-Key CBC MAC. In *Fast Software Encryption*, pages 129–153. Springer, 2003.

[48] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, Internet Engineering Task Force, Oct. 2006.

[49] A. Jungmaier, E. Rescorla, and M. Tuexen. Transport Layer Security over Stream Control Transmission Protocol. RFC 3436, Internet Engineering Task Force, Dec. 2002.

[50] D. Kaminsky. It's The End Of The Cache As We Know It. *Black Hat USA*, 2008.

[51] P. Karn and W. Simpson. ICMP Security Failures Messages. RFC 2521, Internet Engineering Task Force, Mar. 1999.

[52] A. Kato, M. Kanda, and S. Kanno. Camellia Cipher Suites for TLS. RFC 5932, Internet Engineering Task Force, June 2010.

[53] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, Internet Engineering Task Force, Nov. 1998.

[54] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99*, pages 388–397. Springer, 1999.

[55] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO'96*, pages 104–113. Springer, 1996.

[56] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.

[57] H. Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[58] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force, Feb. 1997.

[59] B. Laurie, G. Sisson, R. Arends, and D. Blacka. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155, Internet Engineering Task Force, Mar. 2008.

[60] N. Mavrogiannopoulos. Time is money (in CBC ciphersuites). `http://nmav.gnutls.org/2013/02/time-is-money-for-cbc-ciphersuites.html`.

[61] N. Mavrogiannopoulos. Personal communication, January 2013.

[62] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, Internet Engineering Task Force, Jan. 2008.

[63] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655, Internet Engineering Task Force, July 2012.

[64] D. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). *Submission to NIST*. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf.

[65] P. Mockapetris. Domain Names – Concepts and Facilities. RFC 1034, Internet Engineering Task Force, Nov. 1987.

[66] P. Mockapetris. Domain Names – Implementation and Specification. RFC 1035, Internet Engineering Task Force, Nov. 1987.

[67] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. In *NDSS*. The Internet Society, 2004.

[68] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. Unpublished manuscript, May 2004. http://www.openssl.org/~bodo/tls-cbc.txt.

[69] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, Internet Engineering Task Force, Dec. 1998.

[70] NIST. FIPS 113 – Computer Data Authentication. http://www.itl.nist.gov/fipspubs/fip113.htm.

[71] NIST. FIPS 180-4 – Secure Hash Standard, March 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[72] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286, Internet Engineering Task Force, May 2002.

[73] K. G. Paterson. A Cryptographic Tour of the IPsec Standards. *Information Security Technical Report*, 11(2):72–81, 2006.

[74] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 372–389. Springer, 2011.

[75] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee. WSEC DNS: Protecting Recursive DNS Resolvers from Poisoning Attacks. In *Dependable Systems &*

Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pages 3–12. IEEE, 2009.

[76] L. Poole and V. S. Pai. ConfiDNS: Leveraging Scale and History to Improve DNS Security. In *Proceedings of WORLDS*, volume 6, 2006.

[77] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, Aug. 1980.

[78] J. Postel. Internet Protocol. RFC 791, Internet Engineering Task Force, Sept. 1981.

[79] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, Sept. 1981.

[80] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, Sept. 2001. Updated by RFCs 4301, 6040.

[81] M. Ray. Renegotiating TLS. `https://www.phonefactor.com/sslgapdocs/Renegotiating_TLS.pdf`.

[82] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, Internet Engineering Task Force, Apr. 2006.

[83] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, Internet Engineering Task Force, Jan. 2012.

[84] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, Internet Engineering Task Force, Feb. 2010.

[85] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700, Internet Engineering Task Force, Oct. 1994.

[86] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[87] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, Internet Engineering Task Force, Apr. 1992. Updated by RFC 6151.

[88] P. Rogaway. Problems with Proposed IP Cryptography. Unpublished manuscript, 1995. http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

[89] S. Rose. Applicability Statement: DNS Security (DNSSEC) DNSKEY Algorithm Implementation Status. RFC 6944, Internet Engineering Task Force, Apr. 2013.

[90] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, Internet Engineering Task Force, Aug. 2008.

[91] J. Salowey, T. Petch, R. Gerhards, and H. Feng. Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog. RFC 6012, Internet Engineering Task Force, Oct. 2010.

[92] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, Internet Engineering Task Force, Jan. 2008.

[93] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, Internet Engineering Task Force, Feb. 2012.

[94] S. Sengupta, S. Maitra, G. Paul, and S. Sarkar. RC4: (Non-) Random Words from (Non-)Random Permutations. *IACR Cryptology ePrint Archive*, 2011:448, 2011.

[95] T. Socolofsky and C. Kale. TCP/IP Tutorial. RFC 1180, Internet Engineering Task Force, Jan. 1991.

[96] K. Stamos, G. Pallis, A. Vakali, and M. D. Dikaiakos. Evaluating the Utility of Content Delivery Networks. In *Proceedings of the 4th edition of the UPGRADE-CN workshop on Use of P2P, GRID and agents for the development of content networks*, pages 11–20. ACM, 2009.

[97] Sun, Hung-Min and Chang, Wen-Hsuan and Chang, Shih-Ying and Lin, Yue-Hsun. DependDNS: Dependable Mechanism against DNS Cache Poisoning. In *Cryptology and Network Security*, pages 174–188. Springer, 2009.

[98] J. Touch. Updated Specification of the IPv4 ID Field. RFC 6864, Internet Engineering Task Force, Feb. 2013.

[99] S. Turner and L. Chen. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. RFC 6151, Internet Engineering Task Force, Mar. 2011.

[100] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, Internet Engineering Task Force, Mar. 2011.

[101] A. Vakali and G. Pallis. Content Delivery Networks: Status and Trends. *Internet Computing, IEEE*, 7(6):68–74, 2003.

[102] S. Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology – EUROCRYPT 2002*. Springer, 2002.

[103] P. Vixie, O. Gudmundsson, D. Eastlake 3rd, and B. Wellington. Secret Key Transaction Authentication for DNS (TSIG). RFC 2845, Internet Engineering Task Force, May 2000.

[104] D. Wagner and B. Schneier. Analysis of the SSL 3.0 Protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 29–40, 1996.

[105] X. Wang, A. Yao, and F. Yao. New Collision Search for SHA-1. *Manuscript, presented at rump session of Crypto*, 5, 2005.

[106] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, pages 17–36. Springer, 2005.

[107] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology – EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[108] L. Yuan, K. Kant, P. Mohapatra, and C.-N. Chuah. DoX: A Peer-to-Peer Antidote for DNS Cache Poisoning Attacks. In *ICC*, pages 2345–2350, 2006.

[109] T. Zheng, S. Radhakrishnan, and V. Sarangan. PMAC: An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *IPDPS*, 2005.