Understanding Cryptography through Large-Scale Empiricism

by

David Adrian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2018

Doctoral Committee:

Professor J. Alex Halderman, Chair
Research Professor Peter Honeyman
Associate Professor Chris Peikert
Assistant Professor Florian Schaub

David Adrian

davadria@umich.edu

ACKNOWLEDGMENTS

There are many people to thank; these people will be thanked in the final dissertation. Until then, I'd like to thank my advisor, J. Alex Halderman, for putting up with my unique path into and out of the program, both as a Master's and a Ph.D.ṡtudent. I've learned tremendous amount over the last five years, and I will never forget my time as your student. I'd also like to thank Peter Honeyman for helping push me towards the door, and the rest of my commitee—Chris Peikert and Florian Schaub—for agreeing to be a part of this.

The work in this dissertation was supported in part by the National Science Foundation and the Alfred P. Sloan Foundation, with additional support from the Mozilla Foundation, Supermicro, Google, Cisco, and the Morris Wellman Professorship.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Cryptography is a field born from mathematics, and much of cryptographic research is concerned with proving correctness of both of primitives and protocols, given some set of assumptions. Cryptography is one of the only components of security that can be provably secure. As a result, the cryptography is often considered to be one of the strongest components of security.

Yet historically, cryptography has been one of the most fragile aspects of security. This is not because of issues with the underlying math or the underlying primitives being insecure, but is instead security that is lost in translation between the cryptographers and the implementers.

The process of going from paper to program, or from proof-of-concept to production, introduces mistakes and misunderstandings, which leads to cryptographic failures and insecurity. To address this, the cryptographic research community is beginning to introduce and study new concepts such as misuse-resistant cryptography, and simplified cryptographic APIs. Unfortunately, the state of the art in cryptography engineering remains considerably behind the state of art in research.

To understand where mistakes are being made, and understand where complexity arises from, we must understand how cryptography is actually being used. To better use cryptography to secure the Internet, we must first under how cryptography is being used. Examining CVE databases and trawling through cryptographic code provides one lens into real-world deployments of cryptography, but it does not necessarily reflect the state of the Internet.

Large-scale empirical methods allow us to characterize how cryptography is being used on the Internet, today. Although cryptography is a formal and mathematical science, empirical methods provide additional insight into the fundamental nature of the use of cryptography to secure communication. We have already used empirical methods to make a tangible and measurable impact on the security of the Internet today. We can further use the insights gained from empirical methods to better secure the Internet in the future.

Applying empirical methods to cryptography requires solving engineering challenges. A fundamental methodology underlying the empirical study of cryptography is Internet-wide scanning. While tooling such as ZMap drastically reduced the barrier to entry, Internet-wide scanning still requires additional application-layer tooling to be useful to study cryptography (e.g. the ability to perform large numbers of TLS handshakes), as well as careful experimental design to extract the necessary information from every web server on the Internet, in a systematic, repeatable and consistent method. Data collected from a single scan often approaches one terabyte in size, and correlating across multiple scans may involve processing billions of application-layer handshakes.

In this dissertation I show contributions made to the measurement field itself, and show three different cases where empiricism enabled additional insight when combined with other methods cryptography research.

1.1 Faster Internet-Wide Scanning

I first discuss contributions to the measurement field, which underlies cryptographic empiricisim. There is an abundance of tooling for both active and passive network measurement. It is tempting to assume that the tools required to study cryptography at Internet-scale already exist, however this is often not the case. If the tooling does exist, it often requires herculean effort to answer simple questions, such as "how many HTTPS hosts still support 512-bit key Diffie-Hellman key exchange?". While often overlooked, understanding, contributing, and building tooling is a key aspect of participating in empirical cryptographic research.

As a single example of contributions in this space, in §2, I show improvements to the ZMap scanner which enable it to operate at a full 10Gbps line rate [18]. While ZMap enabled the use of Internet-wide scanning accessible as a measurement method, this work moves towards enabling hourly or real-time measurement of the cryptographic behavior of all Internet-connected systems.

When originally introduced, ZMap was capable of saturating a 1Gbps uplink from a single host, enabling an Internet-wide TCP SYN scan of IPv4 to be performed in forty-five minutes. However, when used with a 10Gbps network interface, ZMap reached barely above the 1Gbps mark. The required thread synchronization during address generation restricted the performance benefit of threading, and limited the ability to leverage multi-core systems. Furthermore, the copy from user space to kernel memory when sending a packet limited total throughput. Scanning a 10Gbps requires sending nearly 15 million packets per second continuously, which allows for only 200 cycles per packet on a 3 GHz system.

I introduced performance improvements to address both of these constraints, and enable ZMap to fully utilize a 10Gbps network link, bringing the total time for a TCP SYN scan of

IPv4 to under five minutes from a single host. While Internet-measurement is often used to provide coarse-grain understanding of the shape of the Internet as a whole, improvements in measurement-collection begin to move the field towards being able to continuously understand the behavior of individual hosts, but at global scale.

1.2 Cost-Effective Global Attacks

Discrete-log and factoring based attacks are generally considered out of reach of attackers. However, informing such attacks with measurement data allows for cost-effective attacks that leverage a single, expensive pre-computation to cheaply attack TLS connections. Furthermore, when combined with broad support for cryptography that was assumed to be dead, these attacks become even cheaper.

Export Cryptography Cryptography has been a regulated as part of the International Traffic in Arms Regulations (ITAR), for decades. In the 1990s, the Department of State chose to implement these regulations by limiting any "exported" cryptography to 40-bits of security for symmetric ciphers, and 512-bits for security for public-key cryptography. Authentication strength (e.g. MAC length), was not regulated. After several court cases started in 1995 surrounding Daniel J. Bernstein's "Snuffle" cryptosystem, the regulations were weakened and moved from the Department of State to the Department of Commerce. The litigation ended in 1999. However, during this time SSLv2 was designed and deprecated, SSLv3 was created and then renamed to TLS and moved into the purview of the IETF, which standardized TLSv1.0 in 1999. All three of these protocols contained compliance mechanisms for "export cryptography".

1.2.1 Attacks on Diffie-Hellman

Diffie-Hellman key exchange is one of the most common public-key cryptographic methods in use in the Internet. In finite field Diffie-Hellman, Alice and Bob agree on a large prime p and an integer g modulo p. Alice chooses a secret integer x_a and transmits a public value $g^{x_a} \bmod p$; Bob chooses a secret integer x_b and transmits his public value $g^{x_b} \bmod p$. Both Alice and Bob can reconstruct a shared secret $g^{x_a x_b} \bmod p$, but the best known way for a passive eavesdropper to reconstruct this secret is to compute the discrete log of either Alice or Bob's public value. Specifically, given g, p, and $g^x \bmod p$, an attacker must calculate x.

We uncovered several weaknesses in how finite-field Diffie-Hellman key exchange was deployed, which drastically affected the cost of decrypting large amounts of TLS traffic. The algorithmic complexity of calculating discrete log is exponential, but the bulk of this computation is dependent solely on p, not the individual secrets a and b chosen by each party. If many hosts use the same groups, then the precomputation cost may be amortized across all the connections across these hosts, rather than requiring core-centuries per observed key exchange. This raise an obvious empirical question: do many hosts share the same set of Diffie-Hellman parameters, and what is the strength of the parameters?

The TLS protocol contains "export-grade" Diffie-Hellman ciphers which use short 512-bit groups. We show that there is a protocol vulnerability in TLS, named Logjam, which allows an attacker who can calculate 512-bit discrete logs to downgrade connections to export-grade Diffie-Hellman ciphers, and decrypt them. If a TLS session were to use 512-bit Diffie-Hellman, it may first appear that individual connections could be broken in 60,000 core-hours, or 120 hours in parallel on commodity hardware. While this is certainly insecure, at first glance it would appear these connections have a small amount forward secrecy, by virtue of using ephemeral Diffie-Hellman key, e.g. each connection would require another 120 hours to be decrypted. This slow process would prohibit active attacks and limit the risk to passive decryption after the fact. This again falls prey to precomputation: if many hosts were to support the same weak parameters, then the computation could be amortized, and individual connections could be broken in real-time, enabling active attacks. In fact, shared sets of parameters is what enables the downgrade. In this case, empirical measurement showed that 80% of vulnerable hosts used the same set of parameters, moving this attack from the theoretical to the practical. While recently there has been a trend towards elliptic curve cryptography, prime-field based Diffie-Hellman remained common in TLS until 2016, when both Firefox and Chrome removed it from their default cipher suites as a result of our work.

1.2.2 Cross-Protocol Attacks

Reusing TLS keys across multiple protocols, such as HTTPS, SMTP, and IMAP, leads to an increased attack surface. Empirical methods allow us to understand the attack surface increase from key reuse. Furthermore, specific vulnerabilities in the TLS protocol and older implementations can be utilized in a cross-protocol context to attack users of a web service without explicitly compromising the private key. This is best shown by the DROWN vulnerability, in which the mere existence of an SSLv2 host that shared a key with a TLS host enabled decryption of otherwise secure TLS connections using modern cryptography.

The DROWN attack further exploited export-grade cryptography with an additional novel insight: Bleichenbacher oracles need not be present in the target protocol under attack, so long as the key is shared between the two protocols. Specifically, DROWN shows how to use protocol vulnerabilities in SSLv2 to attack TLS 1.2. The SSLv2 protocol includes support export symmetric ciphers which are seeded via only five bytes of key material encrypted using RSA PKCS#1 v1.5. The SSLv2 protocol also requires the server to send data to the client that is derived from the shared secret, without first verifying that the client has possession of the secret. When combined with the malleability of RSA, culminates in a Bleichenbacher oracle that can be used to attack TLS 1.2

Beyond DROWN, the TLS protocol has a fundamentally cross-protocol attack surface. X.509 certificates are not bound to any particular protocol or port. Furthermore, even if distinct services, such as mail and web servers, use different keys, so long as they share any name on the certificate, the transport-layer security of all connections to that name are limited to the security of the weakest TLS implementation or configuration. Even traditionally web-based padding oracle attacks, such as POODLE, or the AES-NI padding-oracle in OpenSSL, non-web servers can be exploited by active attackers targeting web users. The attacker can rewrite the TCP connection to an alternative port, and fill-in any pre-handshake protocol dialogue (e.g. by sending an EHLO or STARTTLS command in SMTP). Ignoring vulnerabilities in TLS itself, an unpatched piece of software with a known RCE using the same key as a well-configured and up to date web server places web clients, should the key be stolen via traditional software exploitation. We can place an upper bound on the increased attack surface, by measuring key and name reuse across TLS in different application-layer protocols on different ports.

1.2.3 Weaknesses from Export Cryptography

As shown by Freak, Logjam, and DROWN, the security of TLS and export cryptography are fundamentally linked. Export cryptography is a unique constraint with a fundamentally dangerous goal: weaken cryptography, without weakening cryptography. Internet measurement techniques show us that the export regulations weakened protocol design to the point where the regulations are directly harmful to the security of the Internet today. These empirical techniques show that these attacks are not theoretical, leveraging protocols that have long-since disappeared, but instead are a dark side of backwards compatibility, harming real users today. Although the regulations went out of effect by 1999, the cryptography remains. At their respective times of disclosure, 36.7% of IPv4 HTTPS hosts were vulnerable to FREAK, 4.9% were vulnerable to Logjam, and 26% were vulnerable to DROWN. All forms of export cryptography have been broken: export RSA key exchange was broken by FREAK, export Diffie-Hellman key exchange was broken by Logjam, and export symmetric ciphers were broken by DROWN. In all cases, empirical research enabled the full understanding of the effects and impacts of these issues.

1.3 Fragile Cryptographic Ecosystems

Often, cryptographers are aware of attacks for decades, but there is a knowledge gap between the implementers and researchers. A specific example of this is prime-field based Diffie-Hellman key exchange, in which implementation and hosts often use groups which unnecessarily open up the likelihood of small-subgroup confinement and key recovery attacks [130]. I used empirical techniques provide insight into the Diffie-Hellman group selection at Internet-scale, showing that while a common recommendation is that p should be a "safe" prime such that $p = 2q + 1$ for some prime q, many implementations instead use non-safe "DSA" parameters with potentially unsafe subgroups of order q. Several standards, including NIST SP 800-56A [24] and RFC 5114 [96], advocate the use of these parameters for Diffie-Hellman key exchange, and while it is possible to use such parameters securely, additional validation checks are necessary to prevent small-subgroup attacks.

We measured the prevalence of these parameter choices in the wild for HTTPS, POP3S, SMTP with STARTTLS, SSH, IKEv1, and IKEv2, finding millions of hosts using DSA and other non-"safe" primes for Diffie-Hellman key exchange, many of them in combination with potentially vulnerable behaviors. Beyond simply using DSA primes, small subgroup attacks require a number of complex, special conditions to go wrong in order to be feasible, described in §5. While seems unlikely that any implementation would satisfy enough of these requirements to be vulnerable to an attack, it also seemed unlikely that implementations would use non-safe primes for key exchange in the first place. Empirical methods did not reveal an Internet-wide vulnerability, but rather an Internet-wide case of accidental complexity and fragility. Given the amount of complexity exposed by the underlying cryptographic APIs for Diffie-Hellman, it is remarkable that any implementation was safe. Understanding the root causes of this complexity and confusion enables better protocol design in the future.

CHAPTER 2

Zippier

We introduce optimizations to the ZMap network scanner that achieve a 10-fold increase in maximum scan rate. By parallelizing address generation, introducing an improved blacklisting algorithm, and using zero-copy NIC access, we drive ZMap to nearly the maximum throughput of 10 gigabit Ethernet, almost 15 million probes per second. With these changes, ZMap can comprehensively scan for a single TCP port across the entire public IPv4 address space in 4.5 minutes given adequate upstream bandwidth. We consider the implications of such rapid scanning for both defenders and attackers, and we briefly discuss a range of potential applications.

2.1 Introduction

In August 2013, we released ZMap, an open-source network scanner designed to quickly perform Internet-wide network surveys [49]. From a single machine, ZMap is capable of scanning at 1.44 million packets per second (Mpps), the theoretical limit of gigabit Ethernet. At this speed, ZMap can complete a scan targeting one TCP port across the entire public IPv4 address space in under 45 minutes—a dramatic improvement compared to weeks [49] or months [50] required using Nmap. Yet even at gigabit linespeed, ZMap does not utilize the full bandwidth of the fastest readily available connections: 10 GigE uplinks are now offered by Amazon EC2 [1] and at a growing number of research institutions.

In this paper, we scale ZMap to 10 GigE speeds by introducing a series of performance enhancements. These optimizations allow scanning speeds that provide higher temporal resolution when conducting Internet-wide surveys and make it possible to quickly complete complex multipacket studies.

Scanning at 10 GigE linespeed necessitates sending nearly 15 Mpps continuously. For single-packet probes such as SYN scans, this allows only 200 cycles per probe on a 3 GHz core. An L2 cache miss might incur a cost of almost 100 cycles, so it essential to make efficient use of both CPU and memory. In order to generate and transmit packets at this rate, we introduce modifications that target the three most expensive per-probe operations in ZMap:

1. *Parallelized address generation.* ZMap uses a multiplicative cyclic group to iterate over a random permutation of the address space, but this becomes a bottleneck at multigigabit speeds. We implement a mutex-free sharding mechanism that spreads address generation across multiple threads and cores.

2. *Optimized address constraints.* Responsible scanning requires honoring requests from networks that opt out, but over time this can result in large and complex blacklists. We develop an optimized address constraint data structure that allows ZMap to efficiently cycle through allowed targets.

3. *Zero-copy packet transmission.* ZMap sends Ethernet frames using a raw socket, which avoids the kernel's TCP/IP stack but still incurs a per-packet context switch. We switch to using the PF_RING Zero Copy (ZC) interface, which bypasses the kernel and reduces memory bandwidth.

These enhancements enable ZMap to scan at 14.23 Mpps, 96% of the theoretical limit of 10 GigE. In order to confirm these performance gains, we completed a full scan of the IPv4 address space in 4m29s—to our knowledge, the fastest Internet-wide scan yet reported.

The ability to scan at 10 GigE speeds creates new opportunities for security researchers. It allows for truer snapshots of the state of the Internet by reducing error due to hosts that move or change during the scan. Likewise, it enables more accurate measurement of time-critical phenomena, such as vulnerability patching in the minutes and hours after public disclosure. On the other hand, it raises the possibility that attackers could use 10 GigE to exploit vulnerabilities with alarming speed.

2.2 Related Work

Many network scanning tools have been introduced [49,65,85,94,99], although until recently most were designed for scanning small networks. One of the most popular is Nmap [99], a highly capable network exploration tool. Nmap is well suited for vertical scans of small networks or individual hosts, but the original ZMap implementation outperformed it on horizontal Internet-wide scans by a factor of 1300 [49]. Our enhancements to ZMap improve its performance by another factor of ten.

ZMap is not the first Internet-wide scanner to use PF_RING to send at speeds greater than 1 Gbps. Masscan, released in September 2013, also utilizes PF_RING and claims the ability to scan at 25 Mpps using dual 10 GigE ports—84% of the theoretical limit of dual

10 GigE [65]. We present a more detailed comparison to Masscan in Section 2.4.3. While the Masscan team did not have the facilities to perform live network tests at rates higher than 100,000 pps [65], we report what we believe is the first Internet-wide scan conducted at 10 GigE speeds.

2.3 Performance Optimizations

ZMap achieves this performance based on a series of architectural choices that are geared towards very large, high-speed scans [49]. It avoids per-connection state by embedding tracking information in packet fields that will be echoed by the remote host, using an approach similar to SYN cookies [25]. It eschews timeouts and simplifies flow control by scanning according to a random permutation of the address space. Finally, it avoids the OS's TCP/IP stack and writes raw Ethernet frames.

This architecture allows ZMap to exceed gigabit Ethernet linespeed on commodity hardware, but there are several bottlenecks that prevent it from fully reaching 10 GigE speeds. ZMap's address generation is CPU intensive and requires a global lock, adding significant overhead. Blacklisting ranges of addresses is expensive and scales poorly. Sending each packet requires a context switch and unnecessary copies as packets are passed from userspace to the kernel and then to the NIC [56]. We implement optimizations that reduce each of these bottlenecks.

2.3.1 Address Generation Sharding

Address generation in ZMap is designed to achieve two goals. First, it avoids flooding destination networks by ordering targets according to a pseudorandom permutation of the address space. Second, it enables statistically valid sampling of the address space.

ZMap iterates over a multiplicative group of integers modulo p that represent 32-bit IPv4 addresses. By choosing p to be $2^{32}+15$, the smallest prime larger than 2^{32}, we guarantee that the group $(\mathbb{Z}/p\mathbb{Z})^{\times}$ is cyclic and that it covers the full IPv4 address space. ZMap derives a new random primitive root g for each scan in order to generate new permutation of the address space. The scanner starts at a random initial address a_0 and calculates $a_{i+1} = g \cdot a_i \bmod p$ to iterate through the permutation. The iteration is complete when a_{i+1} equals a_0.

The most expensive part of this scheme is the modulo operation, which must be performed at every step of the iteration. Unfortunately, the modulo operation cannot currently be performed by multiple threads at once, because each address in the permutation is dependent on the previous—calculating the next address requires acquiring a lock over the entire iterator state.

To remove this bottleneck and efficiently distribute address generation over multiple cores, we extend ZMap to support sharding. In the context of ZMap, a shard is a partition of the IPv4 address space that can be iterated over independently from other shards; assigning one shard to each thread allows for independent, mutex-free execution. Each shard contains a disjoint subset of the group, with the union of all the shards covering the entire group.

To define n shards, we choose an initial random address a_0 and assign each sequential address a_j in the permutation to shard $j \bmod n$. To implement this, we initialize shards $1 \ldots n$ with starting addresses $a_0, \ldots, a_{n-1}$, which can be efficiently calculated as $a_0 \cdot g^{0,\ldots,n-1}$. To iterate, we replace g with g^n, which "skips forward" in the permutation by n elements at each step. Each shard computes $a_{i+1} = a_i \cdot g^n \bmod p$ until reaching its shard specific ending address a_{e_j}. For example, if there were three shards, the first would scan $\{a_0,\ a_3 = g^3 \cdot a_0,\ a_6 = g^3 \cdot a_3, \ldots,\ a_{e_1}\}$, second $\{a_1,\ a_4 = g^3 \cdot a_4,\ a_7 = g^3 \cdot a_4, \ldots,\ a_{e_2}\}$, and third $\{a_2,\ a_5 = g^3 \cdot a_0,\ a_8 = g^3 \cdot a_5, \ldots,\ a_{e_3}\}$. We illustrate the process in Figure 2.1.

After pre-calculating the shard parameters, we only need to store three integers per shard: the starting address a_0, the ending address a_e, and the current address a_i. The iteration factor g^n and modulus p are the same for all shards. Each thread can then iterate over a single shard independently of the other threads, and no global lock is needed to determine the next address to scan. Multiple shards can operate within the same ZMap process as threads (the configuration we evaluate in this paper), or they can be split across multiple machines in a distributed scanning mode.

Benchmarks To measure the impact of sharding in isolation from our other enhancements, we conducted a series of scans, each covering a 1% sample of the IP address space, using our local blacklist file and a 10 GigE uplink. Without sharding, the average bandwidth utilization over 10 scans was 1.07 Gbps; with sharding, the average increased to 1.80 Gbps, an improvement of 68%.

2.3.2 Blacklisting and Whitelisting

ZMap address constraints are used to limit scans to specific areas of the network (whitelisting) or to exclude particular address ranges (blacklisting), such as IANA reserved allocations [76]. Blacklisting can also be used to comply with requests from network operators who want to be excluded from receiving probe traffic. Good Internet citizenship demands that ZMap users honor such requests, but after many scans over a prolonged time period, a user's blacklist might contain hundreds of excluded prefixes.

Even with complicated address constraints, ZMap must be able to efficiently determine whether any given IP address should be part of the scan. To support 10 GigE linespeed, we implemented a combination tree- and array-based data structure that can efficiently

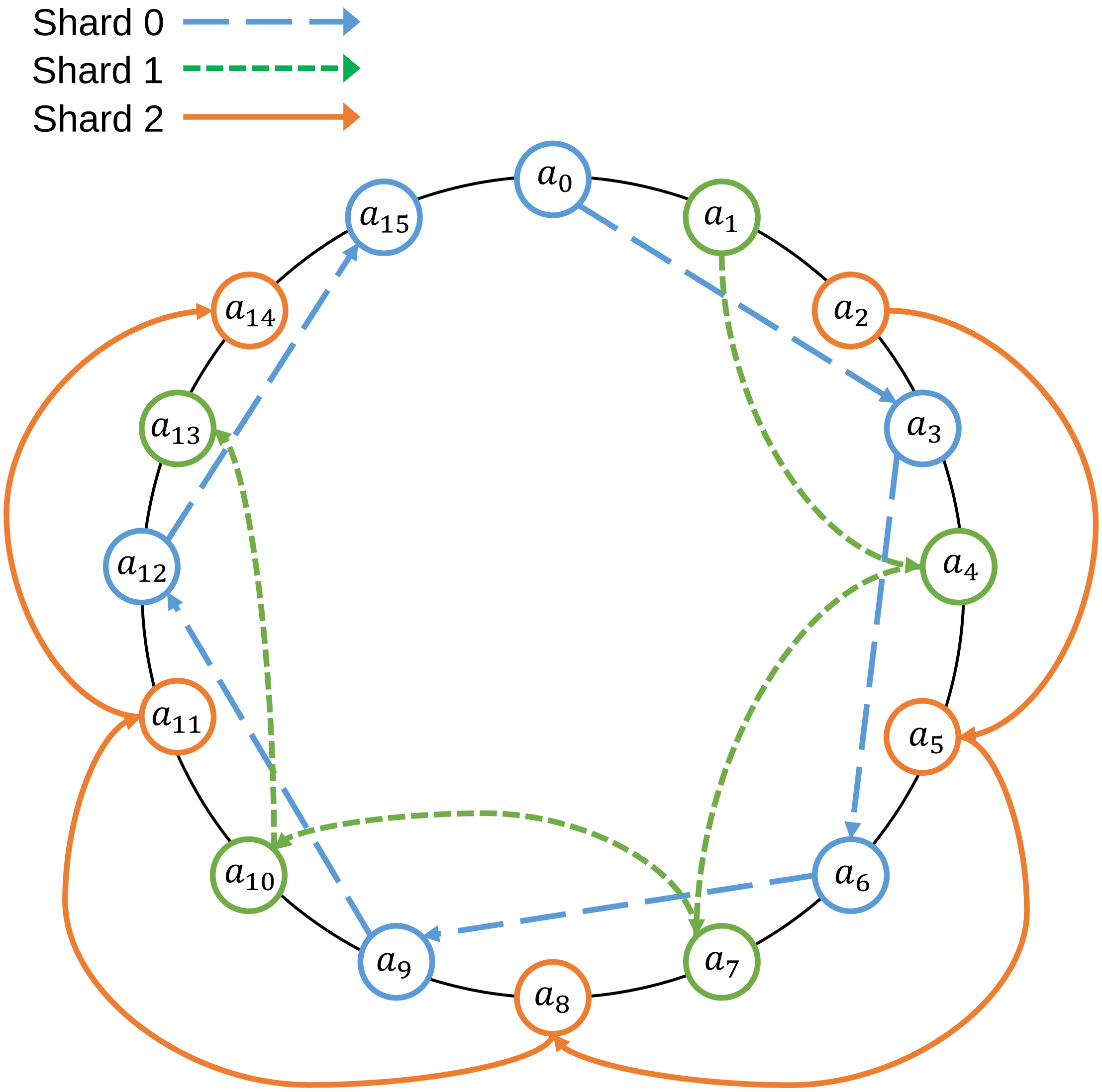

Figure 2.1: **Sharding Visualization** — This is a configuration with three shards ($n = 3$). Shards $0, 1, 2$ are initialized with starting addresses a_0, a_1, a_2. Each arrow represents performing $a_i \cdot g^3$, a step forward by three elements in the permutation.

manipulate and query allowed addresses.

The IPv4 address space is modeled as a binary tree, where each node corresponds to a network prefix. For example, the root represents 0.0.0.0/0, and its children, if present, represent 0.0.0.0/1 and 128.0.0.0/1. Each *leaf* is colored either white or black, depending on whether or not the corresponding prefix is allowed to be scanned. ZMap constructs the tree by sequentially processing whitelist and blacklist entries that specify CIDR prefixes. For each prefix, ZMap sets the color of the corresponding leaf, adding new nodes or pruning the tree as necessary.

Querying whether an address may be scanned involves walking the tree, beginning with the most significant bit of the address, until arriving at a leaf and returning the color. However, a slightly different operation is used during scanning. To make efficient use of the pseudorandom permutation described above, we determine the number of allowed addresses n (which may be much smaller than the address space if a small whitelist is specified) and select a permutation of approximately the same size. We then map from this permutation of $1,\ldots,n$ to allowed addresses $a_1,\ldots,a_n$. Each node in the tree maintains the total number of allowed addresses covered by its descendants, allowing us to efficiently find the ith allowed address using a simple recursive procedure.

As a further optimization, after the tree is constructed, we assemble a list of /20 prefixes that are entirely allowed and reassign the address indices so that these prefixes are ordered before any other allowed addresses. We then use an array of these prefixes to optimize address lookups. If there are m /20 prefixes that are allowed, then the first $m \cdot 2^{12}$ allowed addresses can be returned using only an array lookup, without needing to consult the tree. The /20 size was determined empirically as a trade off between lookup speed and memory usage.

2.3.3 Zero-Copy NIC Access

Despite ZMap's use of raw Ethernet sockets, sending each probe packet is an expensive operation, as it involves a context switch for the `sendto` system call and requires the scan packet to be transferred through kernel space to the NIC [**?**, 121]. Even with our other enhancements, the high cost of these in-kernel operations prevented ZMap from reaching above 2 Gbps. To reduce these costs, we reimplemented ZMap's network functionality using the PF_RING ZC interface [4]. PF_RING ZC allows userspace code to bypass the kernel and have direct "zero-copy" access to the NIC, making it possible to send packets without any context switches or wasted memory bandwidth.

To boost ZMap to 10 GigE speeds, we implemented a new probe transmission architecture on top of PF_RING. This new architecture uses multiple *packet creation* threads that

Scan Rate	**Hit Rate**	**Duration**
1.44 Mpps (≈1 GigE)	1.00	42:08
3.00 Mpps	0.99	20:47
4.00 Mpps	0.97	15:38
14.23 Mpps (≈10 GigE)	0.63	4:29

Table 2.1: **Performance of Internet-wide Scans** — We show the scan rate, the normalized hit rate, and the scan duration (m:s) for complete Internet-wide scans performed with optimized ZMap.

feed into a single *send* thread. We found that using more than one send thread for PF_RING decreased the performance of ZMap, but that a single packet creation thread was not fast enough to reach line speed. By decoupling packet creation from sending, we are able to combine the parallelization benefits of sharding with the speed of PF_RING.

In the original version of ZMap, multiple send threads each generated and sent packets via a thread-specific raw Ethernet socket. We modify thread responsibilities such that each packet creation thread iterates over one address generation shard and generates and queues the packets. In a tight loop, each packet generation loop calculates the next index in the shard, finds the corresponding allowed IP address using the address constraint tree, and creates an addressed packet in the PF_RING ZC driver's memory. The packet is added to a per-thread single-producer, single-consumer packet queue. The send thread reads from each packet queue as packets come available, and sends them over the wire using PF_RING.

To determine the optimal number of packet creation threads, we performed a series of tests, scanning for 50 seconds using 1–6 packet creation threads, and measured the send rate. We find the optimal number of threads corresponds with assigning one per physical core.

2.4 Evaluation

We performed a series of experiments to characterize the behavior of scanning at speeds greater than 1 Gbps. In our test setup, we completed a full scan of the public IPv4 address space in 4m29s on a server with a 10 GigE uplink. However, at full speed the number of scan results (the hit rate) decreased by 37% compared to a scan at 1 Gbps, due to random packet drop. We find that we can scan at speeds of up to 2.7 Gbps before seeing a substantial drop in hit rate.

We performed the following measurements on a Dell PowerEdge R720 with two Intel Xeon E5-2690 2.9 GHz processors (8 physical cores each plus hyper-threading) and 128 GB of memory running Ubuntu 12.04.4 LTS and the 3.2.0-59-generic Linux kernel. We use a single port on a Intel X540-AT2 (rev 01) 10 GigE controller as our scan interface, using the

PF_RING-aware `ixgbe` driver bundled with PF_RING 6.0.1. We configured ZMap to use one send thread, one receive thread, one monitor thread, and five packet creation threads.

We used a 10 GigE network connection at the University of Michigan Computer Science and Engineering division connected directly to the building uplink, an aggregated 2×10 GigE channel. Beyond the 10 GigE connection, the only special network configuration arranged was static IP addresses. We note that ZMap's performance may be different on other networks depending on local congestion and upstream network conditions.

We performed all of our experiments using our local blacklist file. Our blacklist, which eliminates non-routable address space and networks that have requested exclusion from scanning [47], consists of over 1,000 entries of various-sized network blocks. It results in 3.7 billion allowed addresses—with almost all the excluded space consisting of IANA reserved allocations.

2.4.1 Hit-rate vs. Scan-rate

In our original ZMap study, we experimented with various scanning speeds up to gigabit Ethernet line speed (1.44 Mpps) and found no significant effect on the number of results ZMap found [49]. In other words, from our network, ZMap did not appear to miss any results when it ran faster up to gigabit speed.

In order to determine whether hit-rate decreases with speeds higher than 1 Gigabit, we performed 50 second scans at speeds ranging from 0.1–14 Mpps. We performed 3 trials at each scan rate. As can be seen in Figure 2.2, hit-rate begins to drop linearly after 4 Mpps. At 14 Mpps (close to 10 GigE linespeed), the hit rate is 68% of the hit rate for a 1 GigE scan. However, it is not immediately clear why this packet drop is occurring at these higher speeds—are probe packets dropped by the network, responses dropped by the network, or packets dropped on the scan host due to ZMap?

We first investigate whether response packets are being dropped by ZMap or the network. In the original ZMap work, we found that 99% of hosts respond within 1 second [49]. As such, we would expect that after 1 second, there would be negligible responses. However, as can be seen in Figure 2.3, there is an unexpected spike in response packets after sending completes at 50 seconds for scans at 10 and 14 Mpps. This spike likely indicates that response packets are being dropped by our network, NIC, or ZMap, as destination hosts will resend SYN-ACK packets for more than one minute if an ACK or RST packet is not received.

In order to determine whether the drop of response packets is due to ZMap inefficiencies or upstream network congestion, we performed a secondary scan in which we split the probe

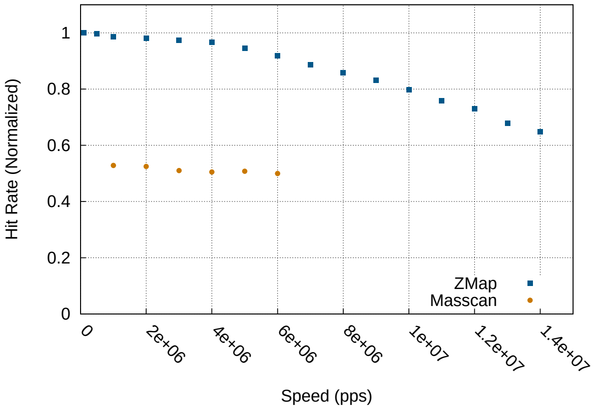

Figure 2.2: **Hit-rate vs. Scan-rate** — ZMap's hit rate is roughly stable up to a scan rate of 4 Mpps, then declines linearly. This drop off may be due to upstreudegrm network congestion. Even using PF_RING, Masscan is unable to achieve scan rates above 6.4 Mpps on the same hardware and has a much lower hit rate.

generation and address processing onto separate machines. The send machine remained the same. The receive machine was an HP ProLiant DL120 G7, with an Intel Xeon E3-1230 processor (4 cores with hyperthreading) and 16 GB of memory, running Ubuntu 12.04.4 LTS and the 3.5.0-52-generic Linux kernel.

As we show in Figure 2.4, this spike does not occur when processing response packets on a secondary server—instead it closely follows the pattern of the slower scans. This indicates that ZMap is locally dropping response packets. However, the split setup received only 4.3% more packets than the single machine—not enough to account for the 31.7% difference between a 14 Mpps and a 1 Mpps scan. If a large number of response packets were dropped due to network congestion, we would not have observed an immediate drop in responses—likely indicating that the root cause of the decreased hit-rate is dropped probe packets.

It is not immediately clear where probe packets are dropped—it is possible that packets are dropped locally by PF_RING, are dropped by local routers due to congestion, or that we are overwhelming destination networks. PF_RING records locally dropped packets, which remained zero throughout our scans, which indicates that packets are not being dropped locally. In order to locate where packet drop is occurring on our network, we calculated the drop rate per AS and found little AS-level correlation for packets dropped by the 10 GigE scans, which suggests that random packet drop is occurring close to our network rather than at particular distant destination networks.

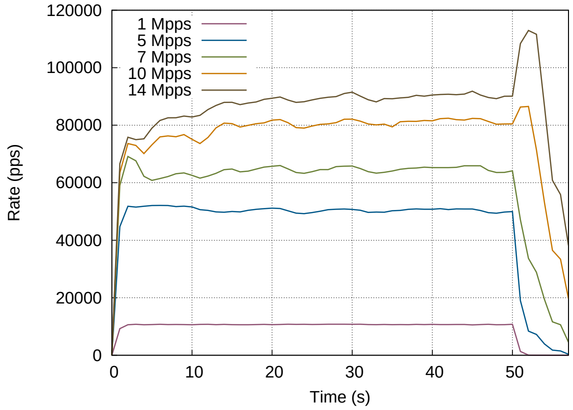

Figure 2.3: **Response Rate During Scans** — This graph shows the rate of incoming SYN-ACKs during 50-second scans. The peaks at the end (after sending finishes) at rates above 7 Mpps indicate that many responses are being dropped and retransmitted before being recorded by ZMap.

2.4.2 Complete Scans

We completed a full Internet-wide scan, allowing ZMap to operate at its full scan rate. This scan achieved an average 14.23 Mpps—96% of the theoretical limit of 10 GigE, completing in 4 minutes, 29 seconds and achieving a hit rate that is 62.5% of that from a 1 GigE scan. We show a comparison to lower speed scans in Table 2.1. As we discussed in the previous section, this decrease is likely due to local network congestion, which results in dropped probe packets. However, more investigation is deserved in order to understand the full dynamics of high-speed scans.

2.4.3 Comparison to Masscan

Masscan advertises the ability to emit probes at 25 Mpps using PF_RING and two 10 GigE adapters, each configured with two RSS queues—84% of linespeed for dual 10 GigE and 166% of linespeed for a single 10 GigE adapter [65]. We benchmarked ZMap and Masscan using the Xeon E3-1230 machine described above. In our experiments, we found that Masscan was able to send at a peak 7.4 Mpps using a single-adapter configuration with two RSS queues, 50% of 10 GigE linespeed. On the same hardware, ZMap is capable of reaching a peak 14.1 Mpps. While Masscan may be able to achieve a higher maximum speed using multiple adapters, ZMap is able to fully saturate a 10 GigE uplink with a single adapter.

Masscan uses a custom Feistel network to "encrypt" a monotonically increasing index to generate a random permutation of the IPv4 address space [64]. While this is computation cheaper than using a cyclic group, this technique results in poor statistical properties,

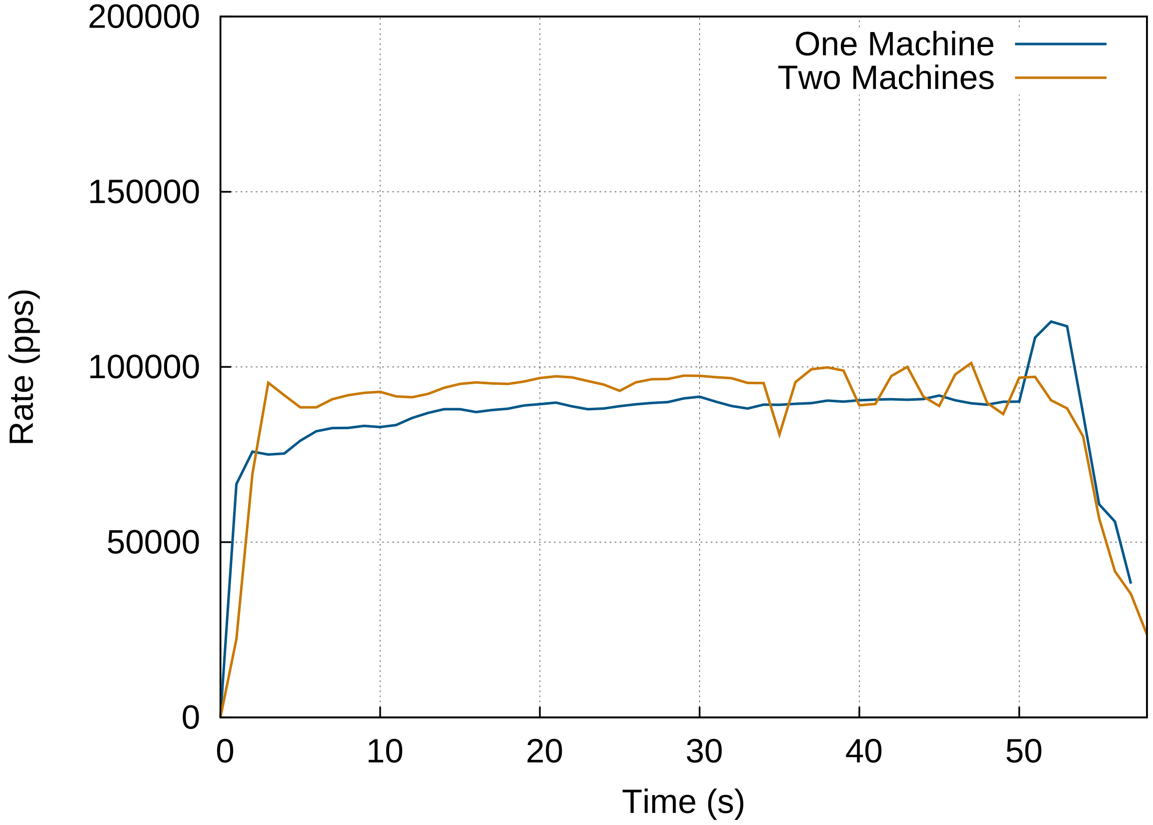

Figure 2.4: **Comparing One and Two Machines** — If we scan at 14 Mpps and use separate machines for the sending and receiving tasks, the spike in the SYN-ACK rate at 50 s disappears, indicating that fewer packets are dropped with the workload spread over two machines. However, overall the two machine configuration received only 4.3% more responses than with one machine, which suggests that network packet loss accounts for the majority of the drop off at higher scan rates.

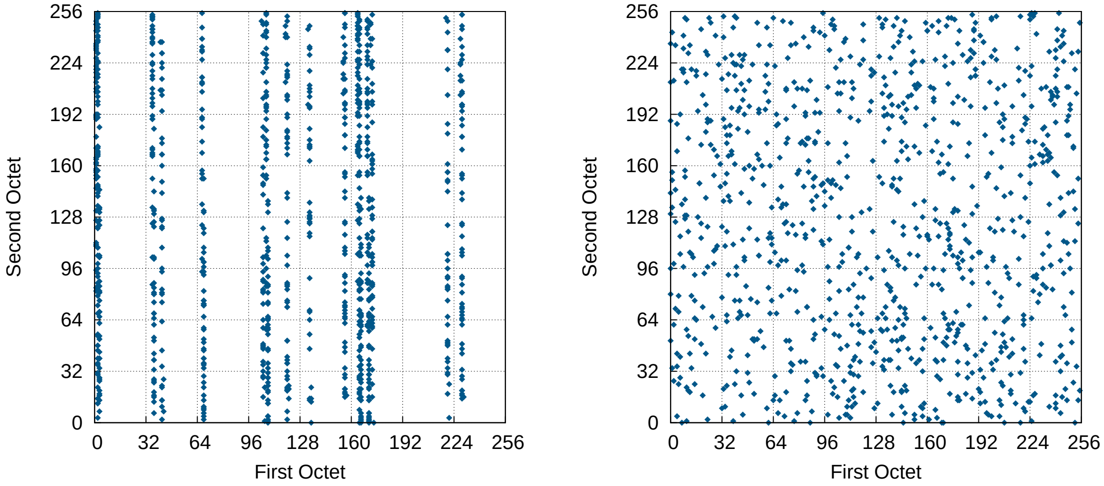

Figure 2.5: **Address Randomization Comparison** — These plots depict the first 1000 addresses of an Internet-wide scan selected by Masscan (*left*) and ZMap (*right*), with the first and second octets mapped to the *x* and *y* coordinates. ZMap's address randomization is CPU intensive but achieves better statistical properties than the cheaper approach used by Masscan, enabling valid sampling. We enhanced ZMap to distribute address generation across multiple cores.

which we show in Figure 2.5. This has two consequences: first, it is not suitable for sampling portions of the address space, and second, there is greater potential for overloading destination networks. This could explain the discrepency in Figure 2.2 if Masscan targeted a less populated subnet.

Masscan and ZMap use a similar sharding approach to parallelize address generation and distribute scans. Both programs "count off" addresses into shards by staggering the offsets of the starting position of each shard within the permutation and iterating a fixed number of steps through each of their permutations. In ZMap, this is implemented by replacing the iteration factor g with g^n. In Masscan, this is simply a matter of incrementing the monotonically increasing index by more than one.

2.5 Applications

In this section, we consider applications that could benefit from 10 GigE scanning and remark on the implications of high-speed scanning for defenders and attackers.

Scanning at faster rates reduces the blur introduced from hosts changing IP addresses by decreasing the number of hosts that may be doubly counted during longer scans. This also increases the ability to discover hosts that are only online briefly. Thus, the ability to complete scans in minutes allows researchers to more accurately create a snapshot of the Internet at a given moment.

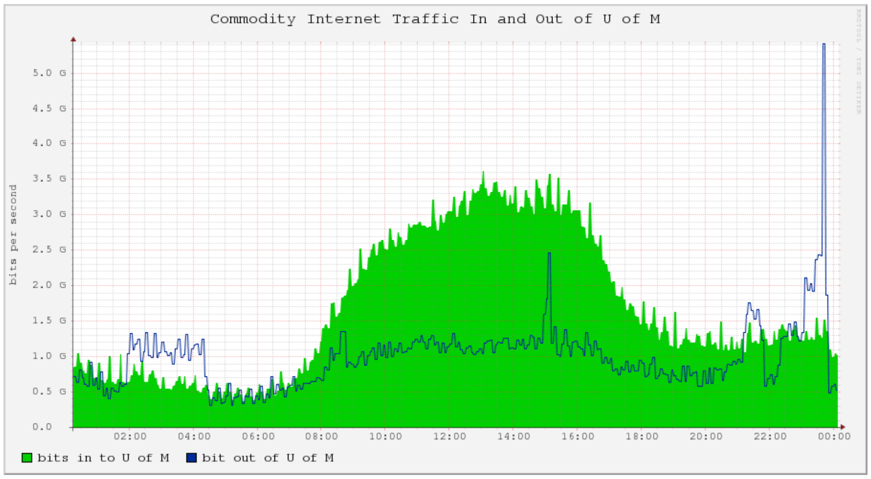

Figure 2.6: **10 GigE Scan Traffic** — An Internet-wide scan at full 10 GigE speed dwarfed all other traffic at the university during this 24 hour period. At 14.23 Mpps, a single machine running ZMap generated 4.6 Gbps in outgoing IP traffic and scanned the entire public IPv4 address space in 4m29s. The massive increase in outbound traffic appears to have caused elevated packet drop. Notable smaller spikes are due to earlier experiments.

Similarly, the increased scan rate enables researchers to complete high-resolution scans when measuring temporal effects. For example, while researchers were able to complete comprehensive scans for the recent Heartbleed Vulnerability every few hours [48], many sites were patched within the first minutes after disclosure. The ability to scan more rapidly could help shed light on patching behavior within this critical initial period.

Faster scan rates also allow for a variety of new scanning-related applications that require multiple packets, including quickly completing global trace routes or performing operating system fingerprinting. Furthermore, the advancement of single-port scanning can be utilized to quickly perform scans of a large number of ports, allowing scanning all privileged ports on a /16 in under 5 seconds and *all* ports in 5 minutes, assuming the attacker has sufficient bandwidth to the target.

The most alarming malicious potential for 10 GigE scanning lies in its ability to find and exploit vulnerabilities *en masse* in a very short time. Durumeric et al. found that attackers began scanning for the Heartbleed vulnerability within 22 hours of its disclosure [48]. While attackers have utilized botnets and worms in order to complete distributed scans for vulnerabilities, recent work [47] has shown that attackers are now also using ZMap, Masscan, and other scanning technology from bullet-proof hosting providers in order to find vulnerable hosts. The increase in scan rates could allow attackers to complete Internet-wide vulnerability scans in minutes as 10 GigE becomes widely available.

2.6 Future Work

We demonstrated that it is possible to perform Internet-wide scans at 10 GigE linespeed, but, at least from our institutional network, we are unable to sustain the expected hit rate as scanning approaches this packet rate. Further investigation is needed to understand this effect and profile ZMap's performance on other networks. One important question is whether the drop off is caused by nearby network bottlenecks (which might be reduced with upgraded network hardware) or whether it arises because such rapid scanning induces congestion on many distant networks—which would represent an inherent limit on scan speed. It is also possible that there are a small number of remote bottlenecks that cause the observed drop in hit rate at high speeds. In that case, identifying, profiling, and removing these bottlenecks could improve performance.

40 GigE hardware currently exists, and 100 GigE is under development [132]. As these networks become more widely available, it may be desirable to optimize and scale Internet-wide scanning to even higher speeds.

2.7 Conclusion

In this work, we introduced enhancements to the ZMap Internet scanner that enable it to scan at up to 14.2 Mpps. The three modifications we present—sharding, optimized address constraints, and integration with PF_RING ZC—enable scanning at close to 10 GigE linespeed. These modifications are available now on experimental ZMap branches and will be merged into mainline ZMap.

With these enhancements, we are able to complete a scan of the public IPv4 address space in 4m29s. However, despite having a well provisioned upstream network, coverage in our experiments drops precipitously when scanning faster than 4 Mpps. While further research is needed to better characterize and reduce the causes of this drop off, it may be related to specific conditions on our network.

As faster network infrastructure becomes more widely available, 10 GigE scanning will enable powerful new applications for both researchers and attackers.

CHAPTER 3

Logjam

We investigate the security of Diffie-Hellman key exchange as used in popular Internet protocols and find it to be less secure than widely believed. First, we present Logjam, a novel flaw in TLS that lets a man-in-the-middle downgrade connections to "export-grade" Diffie-Hellman. To carry out this attack, we implement the number field sieve discrete log algorithm. After a week-long precomputation[1] for a specified 512-bit group, we can compute arbitrary discrete logs in that group in about a minute. We find that 82% of vulnerable servers use a single 512-bit group, allowing us to compromise connections to 7% of Alexa Top Million HTTPS sites. In response, major browsers have changed to reject short groups.

We go on to consider Diffie-Hellman with 768- and 1024-bit groups. We estimate that even in the 1024-bit case, the computations are plausible given nation-state resources. A small number of fixed or standardized groups are used by millions of servers; performing precomputation for a single 1024-bit group would allow passive eavesdropping on 18% of popular HTTPS sites, and a second group would allow decryption of traffic to 66% of IPsec VPNs and 26% of SSH servers. A close reading of published NSA leaks shows that the agency's attacks on VPNs are consistent with having achieved such a break. We conclude that moving to stronger key exchange methods should be a priority for the Internet community.

3.1 Introduction

Diffie-Hellman key exchange is a popular cryptographic algorithm that allows Internet protocols to agree on a shared key and negotiate a secure connection. It is fundamental to protocols such as HTTPS, SSH, IPsec, SMTPS, and other protocols that rely on TLS. Many protocols use Diffie-Hellman to achieve *perfect forward secrecy*, the property that a compromise of the long-term keys used for authentication does not compromise sessions keys for past connections. We examine how Diffie-Hellman is commonly implemented and deployed with common protocols and find that, in practice, it frequently offers less security than widely believed.

There are two reasons for this. First, a surprising number of servers use weak Diffie-Hellman parameters or maintain support for obsolete 1990s-era "export-grade" crypto. More critically, the common practice of using standardized, hard-coded, or widely shared Diffie-Hellman parameters has the effect of dramatically reducing the cost of large-scale attacks, bringing some within range of feasibility today.

The current best technique for attacking Diffie-Hellman relies on compromising one of the private exponents (a, b) by computing the discrete logarithm of the corresponding

[1] Except where otherwise noted, the experimental data and network measurements for this article were obtained in early 2015.

public value ($g^a \bmod p$, $g^b \bmod p$). With state-of-the-art number field sieve algorithms, computing a single discrete log is more difficult than factoring an RSA modulus of the same size. However, an adversary who performs a large precomputation for a prime p can then quickly calculate arbitrary discrete logs in that group, amortizing the cost over all targets that share this parameter. Although this fact is well known among mathematical cryptographers, it seems to have been lost among practitioners deploying cryptosystems. We exploit it to obtain the following results:

Active attacks on export ciphers in TLS We introduce Logjam, a new attack on TLS by which a man-in-the-middle attacker can downgrade a connection to export-grade cryptography. This attack is reminiscent of the FREAK attack [27] but applies to the ephemeral Diffie-Hellman ciphersuites and is a TLS protocol flaw rather than an implementation vulnerability. We present measurements that show that this attack applies to 8.4% of Alexa Top Million HTTPS sites and 3.4% of all HTTPS servers that have browser-trusted certificates.

To exploit this attack, we implemented the number field sieve discrete log algorithm and carried out precomputation for two 512-bit Diffie-Hellman groups used by more than 92% of the vulnerable servers. This allows us to compute individual discrete logs in about a minute. Using our discrete log oracle, we can compromise connections to over 7% of Top Million HTTPS sites. Discrete logs over larger groups have been computed before [33], but, as far as we are aware, this is the first time they have been exploited to expose concrete vulnerabilities in real-world systems.

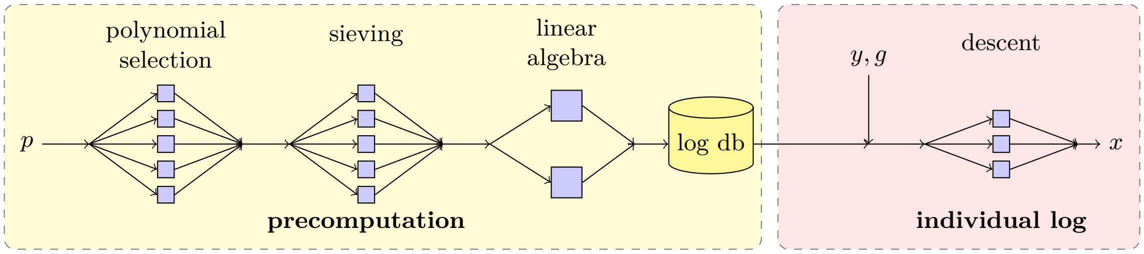

Figure 3.1: **Number field sieve for discrete log** — This algorithm consists of a precomputation stage that depends only on the prime p and a descent stage that computes individual logarithms. With sufficient precomputation, an attacker can quickly break any Diffie-Hellman instances that use a particular p.

Risks from common 1024-bit groups We explore the implications of precomputation attacks for 768- and 1024-bit groups, which are widely used in practice and still considered secure. We estimate the computational resources necessary to compute discrete logs in groups of these sizes, concluding that 768-bit groups are within range of academic teams, and 1024-bit groups may plausibly be within range of nation-state adversaries. In both cases, individual logarithms can be quickly computed after the initial precomputation.

We then examine evidence from published Snowden documents that suggests NSA may already be exploiting 1024-bit Diffie-Hellman to decrypt VPN traffic. We perform measurements to understand the implications of such an attack for popular protocols, finding that an attacker who could perform precomputations for ten 1024-bit groups could passively decrypt traffic to about 66% of IKE VPNs, 26% of SSH servers, and 24% of popular HTTPS sites.

Mitigations and lessons In response to the Logjam attack, mainstream browsers have implemented a more restrictive policy on the size of Diffie-Hellman groups they accept, and Chrome has discontinued support for finite field key exchanges. We further recommend that TLS servers disable export-grade cryptography and carefully vet the Diffie-Hellman groups they use. In the longer term, we advocate that protocols migrate to elliptic curve Diffie-Hellman.

3.2 Diffie-Hellman Cryptanalysis

Diffie-Hellman key exchange was the first published public-key algorithm [43]. In the simple case of prime groups, Alice and Bob agree on a prime p and a generator g of a multiplicative subgroup modulo p. Then each generates a random private exponent, a and b. Alice sends $g^a \bmod p$, Bob sends $g^b \bmod p$, and each computes a shared secret $g^{ab} \bmod p$. While there is also a Diffie-Hellman exchange over elliptic curve groups, we address only the "mod p" case.

The security of Diffie-Hellman is not known to be equivalent to the discrete logarithm problem, but computing discrete logs remains the best known cryptanalytic attack. An attacker who can find the discrete log x from $y = g^x \bmod p$ can easily find the shared secret.

Textbook descriptions of discrete log can be misleading about the computational tradeoffs, for example by optimizing for computing a *single* discrete log. In fact, as illustrated in Figure 3.1, a single large precomputation on p can be used to efficiently break *all* Diffie-Hellman exchanges made with that prime.

Diffie-Hellman is typically implemented with prime fields and large group orders. In this case, the most efficient discrete log algorithm is the number field sieve (NFS) [62, 82, 124]. The algorithm has four stages with different computational properties. The first three steps are only dependent on the prime p and comprise most of the computation.

First is *polynomial selection*, in which one finds a polynomial $f(z)$ defining a number field $\mathbb{Q}[z]/f(z)$ for the computation. This parallelizes well and is only a small portion of the runtime.

In the second stage, *sieving*, one factors ranges of integers and number field elements in batches to find many relations of elements, all of whose prime factors are less than some bound B (called B-smooth). Sieving parallelizes well, but is computationally expensive, because we must search through and attempt to factor many elements.

In the third stage, *linear algebra*, we construct a large, sparse matrix consisting of the coefficient vectors of prime factorizations we have found. This stage can be parallelized in a limited fashion, and produces a database of logarithms which are used as input to the final stage.

The final stage, *descent*, actually deduces the discrete log of the target y. We re-sieve until we find a set of relations that allow us to write the logarithm of y in terms of the logarithms in the precomputed database. Crucially, descent is the only NFS stage that involves y (or g), so polynomial selection, sieving, and linear algebra can be done once for a prime p and reused to compute the discrete logs of many targets.

The numerous parameters of the algorithm allow some flexibility to reduce time on some computational steps at the expense of others. For example, sieving more will result in a smaller matrix, making linear algebra cheaper, and doing more work in the precomputation makes the final descent step easier.

Standard primes Generating safe primes[2] can be computationally burdensome, so many implementations use standardized Diffie-Hellman parameters. A prominent example is the Oakley groups [112], which give “safe” primes of length 768 (Oakley Group 1), 1024 (Oakley Group 2), and 1536 (Oakley Group 5). These groups were published in 1998 and have been used for many applications since, including IKE, SSH, Tor, and OTR.

When primes are of sufficient strength, there seems to be no disadvantage to reusing them. However, widespread reuse of Diffie-Hellman groups can convert attacks that are at the limits of an adversary’s capabilities into devastating breaks, since it allows the attacker to amortize the cost of discrete log precomputation among vast numbers of potential targets.

3.3 Attacking TLS

TLS supports Diffie-Hellman as one of several possible key exchange methods, and prior to public disclosure of the attack, about two-thirds of popular HTTPS sites supported it, most commonly using 1024-bit primes. However, a smaller number of servers also support legacy “export-grade” Diffie-Hellman using 512-bit primes that are well within reach of NFS-based cryptanalysis. Furthermore, for both normal and export-grade Diffie-Hellman, the vast majority of servers use a handful of common groups.

In this section, we exploit these facts to construct a novel attack against TLS, which

[2] An odd prime p is safe when $(p-1)/2$ is prime.

Source	Popularity	Prime

Apache	82%	`9fdb8b8a004544f0045f1737d0ba2e0b`
		`274cdf1a9f588218fb435316a16e3741`
		`71fd19d8d8f37c39bf863fd60e3e3006`
		`80a3030c6e4c3757d08f70e6aa871033`

mod_ssl	10%	`d4bcd52406f69b35994b88de5db89682`
		`c8157f62d8f33633ee5772f11f05ab22`
		`d6b5145b9f241e5acc31ff090a4bc711`
		`48976f76795094e71e7903529f5a824b`

(*others*)	8%	(463 distinct primes)

Table 3.1: **Top 512-bit DH primes for TLS** — 8.4% of Alexa Top 1M HTTPS domains allow DHE_EXPORT, of which 92.3% use one of the two most popular primes, shown here.

we call the Logjam attack. First, we perform NFS precomputations for the two most popular 512-bit primes on the web, so that we can quickly compute the discrete log for any key exchange message that uses one of them. Next, we show how a man-in-the-middle, so armed, can attack connections between popular browsers and any server that allows export-grade Diffie-Hellman, by using a TLS protocol flaw to downgrade the connection to export-strength and then recovering the session key. We find that this attack with our precomputations can compromise connections to about 7.8% of HTTPS servers among Alexa Top Million domains.

3.3.1 TLS and Diffie-Hellman

The TLS handshake begins with a negotiation to determine the crypto algorithms used for the session. The client sends a list of supported ciphersuites (and a random nonce *cr*) within the ClientHello message, where each ciphersuite specifies a key exchange algorithm and other primitives. The server selects a ciphersuite from the client's list and signals its selection in a ServerHello message (containing a random nonce *sr*).

TLS specifies ciphersuites supporting multiple varieties of Diffie-Hellman. Textbook Diffie-Hellman with unrestricted strength is called "ephemeral" Diffie-Hellman, or DHE, and is identified by ciphersuites that begin with `TLS_DHE_*`.[3] In DHE, the server is responsible for selecting the Diffie-Hellman parameters. It chooses a group (p,g), computes g^b, and sends a ServerKeyExchange message containing a signature over the tuple (cr,sr,p,g,g^b) using the long-term signing key from its certificate. The client verifies the signature and responds with a ClientKeyExchange message containing g^a.

[3] New ciphersuites that use elliptic curve Diffie-Hellman (ECDHE) are gaining in popularity, but we focus exclusively on the traditional prime field variety.

To ensure agreement on the negotiation messages, and to prevent downgrade attacks, each party computes the TLS master secret from g^{ab} and calculates a MAC of its view of the handshake transcript. These MACs are exchanged in a pair of Finished messages and verified by the recipients.

Export-grade Diffie-Hellman To comply with 1990s-era U.S. export restrictions on cryptography, SSL 3.0 and TLS 1.0 supported reduced-strength DHE_EXPORT ciphersuites that were restricted to primes no longer than 512 bits. In all other respects, DHE_EXPORT protocol messages are identical to DHE. The relevant export restrictions are no longer in effect, but many servers maintain support for backwards compatibility.

To understand how HTTPS servers in the wild use Diffie-Hellman, we modified the ZMap [49] toolchain to offer DHE and DHE_EXPORT ciphersuites and scanned TCP/443 on both the full public IPv4 address space and the Alexa Top 1M domains. The scans took place in March 2015. Of 539,000 HTTPS sites among Top 1M domains, we found that 68.3% supported DHE and 8.4% supported DHE_EXPORT. Of 14.3 million IPv4 HTTPS servers with browser-trusted certificates, 23.9% supported DHE and 4.9% DHE_EXPORT.

While the TLS protocol allows servers to generate their own Diffie-Hellman parameters, just two 512-bit primes account for 92.3% of Alexa Top 1M domains that support DHE_EXPORT (Table 3.1), and 92.5% of all servers with browser-trusted certificates that support DHE_EXPORT. The most popular 512-bit prime was hard-coded into many versions of Apache; the second most popular is the `mod_ssl` default for DHE_EXPORT.

3.3.2 Active Downgrade to Export-Grade DHE

Given the widespread use of these primes, an attacker with the ability to compute discrete logs in 512-bit groups could efficiently break DHE_EXPORT handshakes for about 8% of Alexa Top 1M HTTPS sites, but modern browsers never negotiate export-grade ciphersuites. To circumvent this, we show how an attacker can downgrade a regular DHE connection to use a DHE_EXPORT group, and thereby break both the confidentiality and integrity of application data.

The attack, which we call Logjam, is depicted in Figure 3.2 and relies on a flaw in the way TLS composes DHE and DHE_EXPORT. When a server selects DHE_EXPORT for a handshake, it proceeds by issuing a signed ServerKeyExchange message containing a 512-bit p_{512}, but the structure of this message is identical to the message sent during standard DHE ciphersuites. Critically, the signed portion of the server's message fails to include any indication of the specific ciphersuite that the server has chosen. Provided that a client offers DHE, an active attacker can rewrite the client's ClientHello to offer a corresponding DHE_EXPORT ciphersuite accepted by the server and remove other cipher-

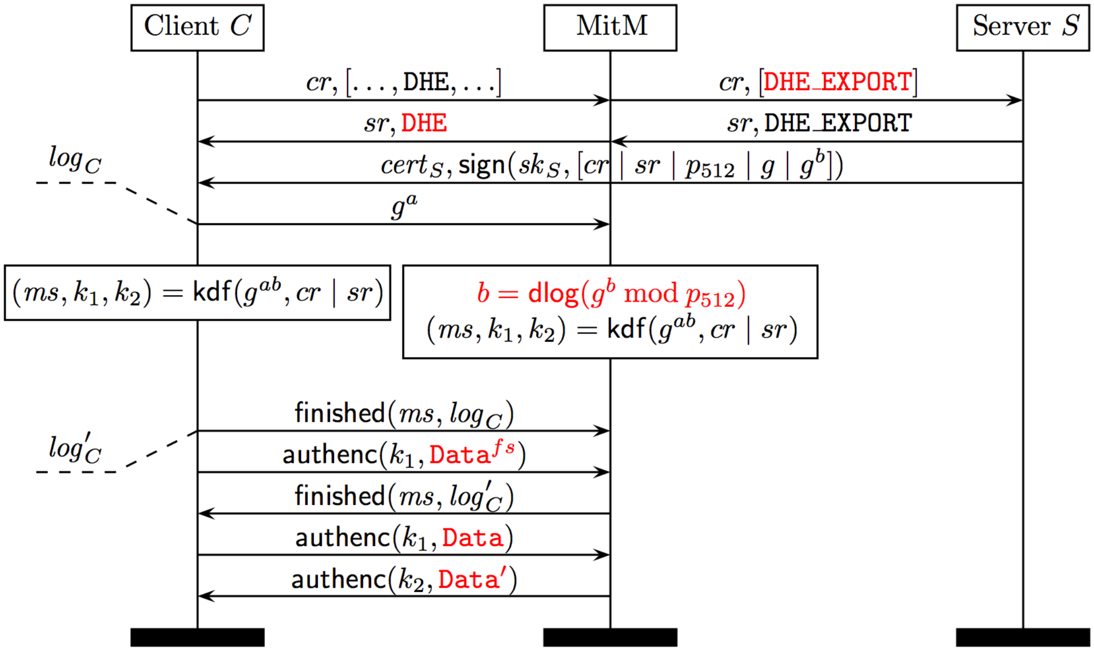
Client C
MitM
Server S
cr, $[\ldots, \mathtt{DHE}, \ldots]$
cr, [DHE_EXPORT]
sr, DHE
sr, DHE_EXPORT
log_C
$cert_S$, $\mathsf{sign}(sk_S, [cr \mid sr \mid p_{512} \mid g \mid g^b])$
g^a
$(ms, k_1, k_2) = \mathsf{kdf}(g^{ab}, cr \mid sr)$
$b = \mathsf{dlog}(g^b \bmod p_{512})$
$(ms, k_1, k_2) = \mathsf{kdf}(g^{ab}, cr \mid sr)$
log'_C
$\mathsf{finished}(ms, log_C)$
$\mathsf{authenc}(k_1, \mathtt{Data}^{fs})$
$\mathsf{finished}(ms, log'_C)$
$\mathsf{authenc}(k_1, \mathtt{Data})$
$\mathsf{authenc}(k_2, \mathtt{Data}')$

Figure 3.2: **The Logjam attack** — A man-in-the-middle can force TLS clients to use export-strength DH with any server that allows DHE_EXPORT. Then, by finding the 512-bit discrete log, the attacker can learn the session key and arbitrarily read or modify the contents. $\mathtt{Data}^{fs}$ refers to False Start application data that some TLS clients send before receiving the server's Finished message.

suites that could be chosen instead. The attacker rewrites the ServerHello response to replace the chosen DHE_EXPORT ciphersuite with a matching non-export ciphersuite and forwards the ServerKeyExchange message to the client as is. The client will interpret the export-grade tuple (p_{512}, g, g^b) as valid DHE parameters chosen by the server and proceed with the handshake. The client and server have different handshake transcripts at this stage, but an attacker who can compute b in close to real time can then derive the master secret and connection keys to complete the handshake with the client.

There are two remaining challenges in implementing this active downgrade attack. The first is to compute individual discrete logs in close to real time, and the second is to delay handshake completion until the discrete log computation has had time to finish.

3.3.3 512-bit Discrete Log Computations

We modified CADO-NFS [127] to implement the number field sieve discrete log algorithm and applied it to the top two DHE_EXPORT primes shown in Table 3.1. Precomputation took 7 days for each prime, after which computing individual logarithms requires a median of 70 seconds.

Precomputation As illustrated in Figure 3.1, the precomputation phase includes the polynomial selection, sieving, and linear algebra steps. For this precomputation, we deliberately sieved more than strictly necessary. This enabled two optimizations: first, with more relations obtained from sieving, we eventually obtain a larger database of known logarithms, which makes the descent faster. Second, more sieving relations also yield a smaller linear algebra step, which is desirable because sieving is much easier to parallelize than linear algebra.

For the polynomial selection and sieving steps, we used idle time on 2000–3000 CPU cores in parallel. Polynomial selection ran for about 3 hours (7,600 core-hours). Sieving ran for 15 hours (21,400 core-hours). This sufficed to collect 40 M relations of which 28 M were unique, involving 15 M primes of at most 27 bits.

From this data set, we obtained a square matrix with 2.2 M rows and columns, with 113 nonzero coefficients per row on average. We solved the corresponding linear system on a 36-node cluster using the block Wiedemann algorithm [39, 128]. Using unoptimized code, the computation finished in 120 hours (60,000 core-hours).

The experiment above was done with CADO-NFS in early 2015. As of 2017, release 2.3 of CADO-NFS [127] performs 20% faster for sieving, and drastically faster for linear algebra, since 9,000 core-hours suffice to solve the same linear system on the same hardware. In total, the wall-clock time for each precomputation was slightly over one week in 2015, and is reduced to about two days with current hardware and more recent software.

Descent Once this precomputation was finished, we were able to run the final descent step to compute individual discrete logs in about a minute. We implemented the descent calculation in a mix of Python and C. On average, computing individual logarithms took about 70 seconds, but the time varied from 34 to 206 seconds on a server with two 18-core Intel Xeon E5-2699 CPUs. For purposes of comparison, a single 512-bit RSA factorization using the CADO-NFS implementation takes about 4 days of wall-clock time on the computer used for the descent [127].

3.3.4 Active Attack Implementation

The main challenge in performing this attack is to compute the shared secret g^{ab} before the handshake completes in order to forge a Finished message from the server. With our descent implementation, the computation takes an average of 70 seconds, but there are several ways an attacker can work around this delay:

Non-browser clients Different TLS clients impose different time limits, after which they kill the connection. Command-line clients such as `curl` and `git` have long or no timeouts, and we can hijack their connections without difficulty.

	Sieving		Linear Algebra		Descent	
	$\log_2 B$	core-years	rows	core-years	core-time	
RSA-512	29	0.3	4.2M	0.03		Timings with default CADO-NFS parameters.
DH-512	27	2.5	2.2M	1.1	10 mins	For the computations in this paper; may be suboptima
RSA-768	37	800	250M	100		Est. based on [90] with less sieving.
DH-768	36	4,000	24M	920	43 hours	Data from [91, Table 1].
RSA-1024	42	≈1,000,000	≈8.7B	≈120,000		Crude estimate based on complexity formula.
DH-1024	40	≈5,000,000	≈0.8B	≈1,100,000	30 days	Crude estimate based on formula and our experiments

Table 3.2: **Estimating costs for factoring and discrete log**. For sieving, we give one important parameter, which is the number of bits of the smoothness bound B. For linear algebra, all costs for DH are for safe primes; for DSA primes with q of 160 bits, this should be divided by 6.4 for 1024 bits, 4.8 for 768 bits, and 3.2 for 512 bits.

TLS warning alerts Web browsers tend to have shorter timeouts, but we can keep their connections alive by sending TLS warning alerts, which are ignored by the browser but reset the handshake timer. For example, this allows us to keep Firefox TLS connections alive indefinitely.

Ephemeral key caching Many TLS servers do not use a fresh value b for each connection, but instead compute g^b once and reuse it for multiple negotiations. For example, F5 BIG-IP load balancers will reuse g^b by default. Microsoft Schannel caches g^b for two hours—this setting is hard-coded. For these servers, an attacker can compute the discrete log of g^b from one connection and use it to attack later handshakes.

TLS False Start Even when clients enforce shorter timeouts and servers do not reuse values for b, the attacker can still break the confidentiality of user requests that use TLS False Start. Recent versions of Chrome, Internet Explorer, and Firefox implement False Start, but their policies on when to enable it vary. Firefox 35, Chrome 41, and Internet Explorer (Windows 10) send False Start data with DHE. In these cases, a man-in-the-middle can record the handshake and decrypt the False Start payload at leisure.

3.4 Nation-State Threats to DH

The previous sections demonstrate the existence of practical attacks against Diffie-Hellman key exchange as currently used by TLS. However, these attacks rely on the ability to downgrade connections to export-grade crypto. In this section we address the following question: how secure is Diffie-Hellman in broader practice, as used in other protocols that do not suffer from downgrade, and when applied with stronger groups?

To answer this question we must first examine how the number field sieve for discrete log scales to 768- and 1024-bit groups. As we argue below, 768-bit groups in relatively widespread use are now within reach for academic computational resources. Additionally, performing precomputations for a small number of 1024-bit groups is plausibly within the resources of nation-state adversaries. The precomputation would likely require special-purpose hardware, but would not require any major algorithmic improvements. In light of these results, we examine several standard Internet security protocols—IKE, SSH, and TLS—to determine their vulnerability. Although the cost of the precomputation for a 1024-bit group is several times higher than for an RSA key of equal size, a one-time investment could be used to attack millions of hosts, due to widespread reuse of the most common Diffie-Hellman parameters. Finally, we apply this new understanding to a set of recently published documents to evaluate the hypothesis that the National Security Agency has *already* implemented such a capability.

3.4.1 Scaling NFS to 768- and 1024-bit DH

Estimating the cost for discrete log cryptanalysis at larger key sizes is far from straightforward due to the complexity of parameter tuning. We attempt estimates up to 1024-bit discrete log based on the existing literature and our own experiments but further work is needed for greater confidence. We summarize all the costs, measured or estimated, in Table 3.2.

DH-768: Completed in 2016 At the time of disclosure, the latest discrete log record was a 596-bit computation. Based on that work, and on prior experience with the 768-bit factorization record in 2009 [90], we made the conservative prediction that it was possible, as explained in §3.2, to put more computational effort into sieving for the discrete log case than for factoring, so that the linear algebra step would run on a slightly smaller matrix. This led to a runtime estimate of around 35,000 core-years, most of which was spent on linear algebra.

This estimate turned out be overly conservative, for several reasons. First, there have been significant improvements in our software implementation (see §3.3.3). In addition, our estimate did not use the Joux-Lercier alternative polynomial selection method [82, §2.1], which is specific to discrete logs. For 768-bit discrete logs, this polynomial selection method leads to a significantly smaller computational cost.

In 2016, Kleinjung et al. completed a 768-bit discrete log computation [91]. While this is a massive computation on the academic scale, a computation of this size has likely been within reach of nation-states for more than a decade. This data is mentioned in Table 3.2.

DH-1024: Plausible with nation-state resources Experimentally extrapolating sieving parameters to the 1024-bit case is difficult due to the tradeoffs between the steps of the algorithm and their relative parallelism. The prior work proposing parameters for factoring a 1024-bit RSA key is thin and we resort to extrapolating from asymptotic complexity. For the number field sieve, the complexity is $\exp\left((k+o(1))(\log N)^{1/3}(\log\log N)^{2/3}\right)$, where N is the integer to factor or the prime modulus for discrete log and k is an algorithm-specific constant. This formula is inherently imprecise, since the $o(1)$ in the exponent can hide polynomial factors. This complexity formula, with $k = 1.923$, describes the overall time for both discrete log and factorization, which are both dominated by sieving and linear algebra in the precomputation. Evaluating the formula for 768- and 1024-bit N gives us estimated multiplicative factors by which time and space will increase from the 768- to the 1024-bit case.

For 1024-bit precomputation, the total time complexity can be expected to increase by a factor of 1220 using the complexity formula, while space complexity increases by its square root, approximately 35. These ratios are relevant for both factorization and discrete log since they have the same asymptotic behavior. For DH-1024, we get a total cost estimate for the precomputation of about 6M core-years.

The time complexity for each individual log after the precomputation should be multiplied by $L_{2^{1024}}(1.206)/L_{2^{768}}(1.206) \approx 86$, where $k = 1.206$ follows from [54]. This last number does not correspond to what we observed in practice and we attribute that to the fact that the descent step has been far less studied.

In practice, it is not uncommon for estimates based merely on the complexity formula to be off by a factor of 10. Estimates of Table 3.2 must therefore be considered with due caution.

For 1024-bit descent, we experimented with our early-abort implementation to inform our estimates for descent initialization, which should dominate the individual discrete log computation. For a random target in Oakley Group 2, initialization took 22 core-days, and yielded a few primes of at most 130 bits to be descended further. In twice this time, we reached primes of about 110 bits. At this point, we were certain to have bootstrapped the descent and could continue down to the smoothness bound in a few more core-days if proper sieving software were available. Thus we estimate that a 1024-bit descent would take about 30 core-days, once again easily parallelizable.

Costs in hardware Although several millions of core-years is a massive computational effort, it is not necessarily out of reach for a nation-state. At this scale, significant cost savings could be realized by developing application-specific hardware given that sieving is a natural target for hardware implementation. To our knowledge, the best prior description of an ASIC implementation of 1024-bit sieving is the 2007 work of Geiselmann and Steinwandt [59]. Updating their estimates for modern techniques and adjusting parameters for discrete log allows us to extrapolate the financial and time costs.

We increase their chip count by a factor of ten to sieve more and save on linear algebra as above giving an estimate of 3M chips to complete sieving in one year. Shrinking the dies from the 130 nm technology node used in the paper to a more modern size reduces costs as transistors are cheaper at newer technologies. With standard transistor costs and utilization, it would cost about $2 per chip to manufacture after fixed design and tape-out costs of roughly $2M [98]. This suggests that an $8M investment would buy enough ASICs to complete the DH-1024 sieving precomputation in one year. Since a step of descent uses sieving, the same hardware could likely be reused to speed calculations of individual logarithms.

Estimating the financial cost for the linear algebra is more difficult since there has been little work on designing chips that are suitable for the larger fields involved in discrete log. To derive a rough estimate, we can begin with general purpose hardware and the core-year estimate from Table 3.2. Using the 300,000 CPU core Titan supercomputer it would take 4 years to complete the 1024-bit linear algebra stage (notwithstanding the fact that estimates from Table 3.2 are known to be extremely coarse, and could be optimistic by a factor of maybe 10). Titan was constructed in 2012 for $94M, suggesting a cost of $0.5B in supercomputers to finish this step in a year. In the context of factorization, moving linear algebra from general purpose CPUs to ASICs has been estimated to reduce costs by a factor of 80 [58]. If we optimistically assume that a similar reduction can be achieved for discrete log, the hardware cost to perform the linear algebra for DH-1024 in one year is plausibly on the order of tens of millions of dollars.

To put this dollar figure in context, the FY 2012 budget for the U.S. Consolidated Cryptologic Program (which includes the NSA) was $10.5 billion[4] [142]. The 2013 budget request, which prioritized investment in “groundbreaking cryptanalytic capabilities to defeat adversarial cryptography and exploit internet traffic” included notable $100M+ increases in two programs under Cryptanalysis & Exploitation Services: “Cryptanalytic IT Systems” (to $247M), and the cryptically named “PEO Program C” (to $360M) [142].

3.4.2 Is NSA Breaking 1024-bit DH?

Our calculations suggest that it is plausibly within NSA’s resources to have performed number field sieve precomputations for a small number of 1024-bit Diffie-Hellman groups. This would allow them to break any key exchanges made with those groups in close to real time. If true, this would answer one of the major cryptographic questions raised by the Edward Snowden leaks: How is NSA defeating the encryption for widely used VPN

[4] The National Science Foundation’s budget was $7 billion.

protocols?

Virtual private networks (VPNs) are widely used for tunneling business or personal traffic across potentially hostile networks. We focus on the Internet Protocol Security (IPsec) VPN protocol using the Internet Key Exchange (IKE) protocol for key establishment and parameter negotiation and the Encapsulating Security Payload (ESP) protocol for protecting packet contents.

IKE There are two versions, IKEv1 and IKEv2, which differ in message structure but are conceptually similar. For the sake of brevity, we will use IKEv1 terminology [88].

Each IKE session begins with a Phase 1 handshake in which the client and server select a Diffie-Hellman group from a small set of standardized parameters and perform a key exchange to establish a shared secret. The shared secret is combined with other cleartext values transmitted by each side, such as nonces and cookies, to derive a value called SKEYID. When authenticated with a pre-shared key (PSK) in IKEv1, the PSK value is incorporated into the derivation of SKEYID.

The resulting SKEYID is used to encrypt and authenticate a Phase 2 handshake. Phase 2 establishes the parameters and key material, KEYMAT, for protecting the subsequently tunneled traffic. Ultimately, KEYMAT is derived from SKEYID, additional nonces, and the result of an optional Phase 2 Diffie-Hellman exchange.

NSA's VPN exploitation process Documents published by Der Spiegel describe NSA's ability to decrypt VPN traffic using passive eavesdropping and without message injection or man-in-the-middle attacks on IPsec or IKE. Figure 3.3 illustrates the flow of information required to decrypt the tunneled traffic.

When the IKE/ESP messages of a VPN of interest are collected, the IKE messages and a small amount of ESP traffic are sent to the Cryptanalysis and Exploitation Services (CES) [141, 143, 146]. Within the CES enclave, a specialized "attack orchestrator" attempts to recover the ESP decryption key with assistance from high-performance computing resources as well as a database of known PSKs ("CORALREEF") [141, 143, 146]. If the recovery was successful, the decryption key is returned from CES and used to decrypt the buffered ESP traffic such that the encapsulated content can be processed [141, 145].

Evidence for a discrete log attack The ability to decrypt VPN traffic does not necessarily indicate a defeat of Diffie-Hellman. There are, however, several features of the described exploitation process that support this hypothesis.

The IKE protocol has been extensively analyzed [36, 103] and is not believed to be exploitable in standard configurations under passive eavesdropping attacks. Absent a vulnerability in the key derivation function or transport encryption, the attacker must recover

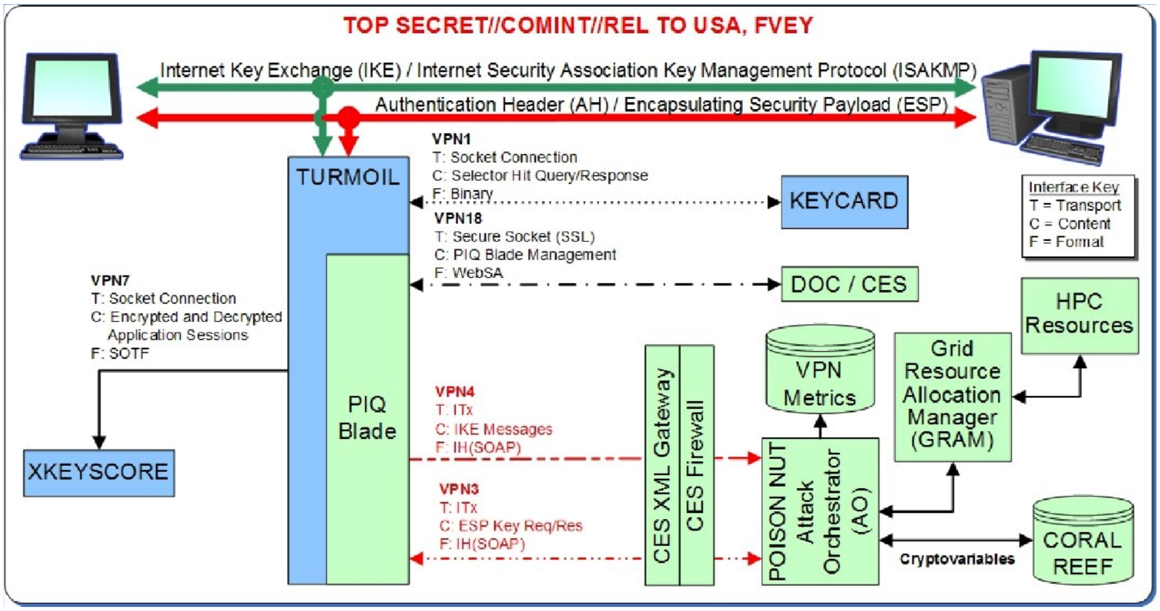

Figure 3.3: **NSA's VPN decryption infrastructure** — This classified illustration published by Der Spiegel [146] shows captured IKE handshake messages being passed to a high-performance computing system, which returns the symmetric keys for ESP session traffic. The details of this attack are consistent with an efficient break for 1024-bit Diffie-Hellman.

the decryption keys. This requires the attacker to calculate SKEYID generated from the Phase 1 Diffie-Hellman shared secret after passively observing an IKE handshake.

While IKE is designed to support a range of Diffie-Hellman groups, our Internet-wide scans (§3.4.3) show that the vast majority of IKE endpoints select one particular 1024-bit DH group even when offered stronger groups. Conducting an expensive, but feasible, precomputation for this single 1024-bit group (Oakley Group 2) would allow the efficient recovery of a large number of Diffie-Hellman shared secrets used to derive SKEYID and the subsequent KEYMAT.

Given an efficient oracle for solving the discrete logarithm problem, attacks on IKE are possible provided that the attacker can obtain the following: (1) a complete two-sided IKE transcript, and (2) any PSK used for deriving SKEYID in IKEv1. The available documents describe both of these as explicit prerequisites for the VPN exploitation process outlined above and provide the reader with internal resources available to meet these prerequisites [143].

Of course, this explanation is not dispositive and the possibility remains that NSA could defeat VPN encryption using alternative means. A published NSA document refers to the use of a router "implant" to allow decryption of IPsec traffic indicating that the use of targeted malware is possible. This implant "allows passive exploitation with just ESP" [143] without the prerequisite of collecting the IKE handshake messages. This indicates that it is

	Vulnerable servers, if the attacker can precompute for . .			
	all 512-bit groups	all 768-bit groups	one 1024-bit group	ten 10
HTTPS Top 1M w/ active downgrade	45,100 (8.4%)	45,100 (8.4%)	205,000 (37.1%)	309
HTTPS Top 1M	118 (0.0%)	407 (0.1%)	98,500 (17.9%)	132
HTTPS Trusted w/ active downgrade	489,000 (3.4%)	556,000 (3.9%)	1,840,000 (12.8%)	3,410
HTTPS Trusted	1,000 (0.0%)	46,700 (0.3%)	939,000 (6.56%)	1,430
IKEv1 IPv4	–	64,700 (2.6%)	1,690,000 (66.1%)	1,690
IKEv2 IPv4	–	66,000 (5.8%)	726,000 (63.9%)	726
SSH IPv4	–	–	3,600,000 (25.7%)	3,600

Table 3.3: **Estimated impact of Diffie-Hellman attacks in early 2015.** We used Internet-wide scanning to estimate the number of real-world servers for which typical connections could be compromised by attackers with various levels of computational resources. For HTTPS, we provide figures with and without downgrade attacks on the chosen ciphersuite. All others are passive attacks.

an alternative mechanism to the attack described above.

The most compelling argument for a pure cryptographic attack is the generality of the NSA's VPN exploitation process. This process appears to be applicable across a broad swath of VPNs without regard to endpoint's identity or the ability to compromise individual endpoints.

3.4.3 Effects of a 1024-bit Break

In this section, we use Internet-wide scanning to assess the impact of a hypothetical DH-1024 break on IKE, SSH, and HTTPS. Our measurements, performed in early 2015, indicate that these protocols would be subject to widespread compromise by a nation-state attacker who had the resources to invest in precomputation for a small number of 1024-bit groups.

IKE We measured how IPsec VPNs use Diffie-Hellman in practice by scanning a 1% random sample of the public IPv4 address space for IKEv1 and IKEv2 (the protocols used to initiate an IPsec VPN connection) in May 2015. We used the ZMap UDP probe module to measure support for Oakley Groups 1 and 2 (two popular 768- and 1024-bit, built-in groups) and which group servers prefer. Of the 80K hosts that responded with a valid IKE packet, 44.2% were willing to negotiate a connection using one of the two groups. We found that 31.8% of IKEv1 and 19.7% of IKEv2 servers support Oakley Group 1 (768-bit) while 86.1% and 91.0% respectively supported Oakley Group 2 (1024-bit). In our sample of IKEv1 servers, 2.6% of profiled servers preferred the 768-bit Oakley Group 1 and 66.1% preferred the 1024-bit Oakley Group 2. For IKEv2, 5.8% of profiled servers chose Oakley Group 1, and 63.9% chose Oakley Group 2.

SSH All SSH handshakes complete either a finite field or elliptic curve Diffie-Hellman exchange. The protocol explicitly defines support for Oakley Group 2 (1024-bit) and Oakley Group 14 (2048-bit) but also allows a server-defined group to be negotiated. We scanned 1% random samples of the public IPv4 address space in April 2015. We find that 98.9% of SSH servers support the 1024-bit Oakley Group 2, 77.6% support the 2048-bit Oakley Group 14, and 68.7% support a server-defined group.

During the SSH handshake, the server selects the client's highest priority mutually supported key exchange algorithm. To estimate what servers will prefer in practice, we performed a scan in which we mimicked the algorithms offered by OpenSSH 6.6.1p1, the latest version of OpenSSH. In this scan, 21.8% of servers preferred the 1024-bit Oakley Group 2, and 37.4% preferred a server-defined group. 10% of the server-defined groups were 1024-bit, but, of those, nearly all provided Oakley Group 2 rather than a custom group.

Combining these equivalent choices, we find that a nation-state adversary who performed NFS precomputations for the 1024-bit Oakley Group 2 could passively eavesdrop on connections to 3.6M (25.7%) publicly accessible SSH servers.

HTTPS DHE is commonly deployed on web servers. 68.3% of Alexa Top 1M sites support DHE, as do 23.9% of sites with browser-trusted certificates. Of the Top 1M sites that support DHE, 84% use a 1024-bit or smaller group, with 94% of these using one of five groups.

Despite widespread support for DHE, a passive eavesdropper can only decrypt connections that organically agree to use Diffie-Hellman. We estimate the number of sites for which this will occur by offering the same sets of ciphersuites as Chrome, Firefox, and Safari. Approximately 24.0% of browser connections with HTTPS-enabled Top 1M sites (and 10% with browser-trusted sites) will negotiate DHE with one of the ten most popular 1024-bit primes; 17.9% of connections with Top 1M sites could be passively eavesdropped given the precomputation for a single 1024-bit prime.

3.5 Recommendations

In this section, we present concrete recommendations to recover the expected security of Diffie-Hellman.

Transition to elliptic curves. Transitioning to elliptic curve Diffie-Hellman (ECDH) key exchange avoids all known feasible cryptanalytic attacks. Current elliptic curve discrete log algorithms do not gain as much of an advantage from precomputation. In addition, ECDH keys are shorter and computations are faster. We recommend transitioning to elliptic curves; this is the most effective solution to the vulnerabilities in this paper. We note that in August 2015, the NSA announced that it was planning to transition away from elliptic curve cryptography for its Suite B cryptographic algorithms and would replace them with algorithms resistant to quantum computers [111]. However, since no fully vetted and standardized quantum-resistant algorithms exist currently, elliptic curves remain the most secure choice for public key operations.

Increase minimum key strengths. To protect against the Logjam attack, server operators should disable DHE_EXPORT and configure DHE ciphersuites to use primes of 2048 bits or larger. Browsers and clients should raise the minimum accepted size for Diffie-Hellman groups to at least 1024 bits in order to avoid downgrade attacks.

Don't deliberately weaken crypto. The Logjam attack illustrates the fragility of cryptographic "front doors". Although the key sizes originally used in DHE_EXPORT were intended to be tractable only to NSA, two decades of algorithmic and computational improvements have significantly lowered the bar to attacks on such key sizes. Despite the eventual relaxation of crypto export restrictions and subsequent attempts to remove support for DHE_EXPORT, the technical debt induced by the additional complexity has left implementations vulnerable for decades. Like FREAK [27], our attacks warn of the long-term debilitating effects of deliberately weakening cryptography.

3.6 Conclusion

We find that Diffie-Hellman key exchange, as used in practice, is often less secure than widely believed. The problems stem from the fact that the number field sieve for discrete log allows an attacker to perform a single precomputation that depends only on the group, after which computing individual logarithms in that group has a far lower cost. Although this is well known to cryptographers, it apparently has not been widely understood by system builders. Likewise, many cryptographers did not appreciate that a large fraction of Internet communication depends on a few small, widely shared groups.

A key lesson is that cryptographers and creators of practical systems need to work together more effectively. System builders should take responsibility for being aware of applicable cryptanalytic attacks. Cryptographers should involve themselves in how crypto is actually being applied, such as through engagement with standards efforts and software review. Bridging the perilous gap that separates these communities will be essential for keeping future systems secure.

CHAPTER 4

Drown

We present DROWN, a novel cross-protocol attack on TLS that uses a server supporting SSLv2 as an oracle to decrypt modern TLS connections.

We introduce two versions of the attack. The more general form exploits multiple unnoticed protocol flaws in SSLv2 to develop a new and stronger variant of the Bleichenbacher RSA padding-oracle attack. To decrypt a 2048-bit RSA TLS ciphertext, an attacker must observe 1,000 TLS handshakes, initiate 40,000 SSLv2 connections, and perform 2^{50} offline work. The victim client never initiates SSLv2 connections. We implemented the attack and can decrypt a TLS 1.2 handshake using 2048-bit RSA in under 8 hours, at a cost of $440 on Amazon EC2. Using Internet-wide scans, we find that 33% of all HTTPS servers and 22% of those with browser-trusted certificates are vulnerable to this protocol-level attack due to widespread key and certificate reuse.

For an even cheaper attack, we apply our new techniques together with a newly discovered vulnerability in OpenSSL that was present in releases from 1998 to early 2015. Given an unpatched SSLv2 server to use as an oracle, we can decrypt a TLS ciphertext in one minute on a single CPU—fast enough to enable man-in-the-middle attacks against modern browsers. We find that 26% of HTTPS servers are vulnerable to this attack.

We further observe that the QUIC protocol is vulnerable to a variant of our attack that allows an attacker to impersonate a server indefinitely after performing as few as 2^{17} SSLv2 connections and 2^{58} offline work.

We conclude that SSLv2 is not only weak, but actively harmful to the TLS ecosystem.

4.1 Introduction

TLS [42] is one of the main protocols responsible for transport security on the modern Internet. TLS and its precursor SSLv3 have been the target of a large number of cryptographic attacks in the research community, both on popular implementations and the protocol itself [104]. Prominent recent examples include attacks on outdated or deliberately weakened encryption in RC4 [21], RSA [28], and Diffie-Hellman [17], different side channels including Lucky13 [20], BEAST [44], and POODLE [107], and several attacks on invalid TLS protocol flows [28, 30, 41].

Comparatively little attention has been paid to the SSLv2 protocol, likely because the known attacks are so devastating and the protocol has long been considered obsolete. Wagner and Schneier wrote in 1996 that their attacks on SSLv2 "will be irrelevant in the long term when servers stop accepting SSL 2.0 connections" [133]. Most modern TLS clients do not support SSLv2 at all. Yet in 2016, our Internet-wide scans find that out of 36 million HTTPS servers, 6 million (17%) support SSLv2.

A Bleichenbacher attack on SSLv2. Bleichenbacher's padding oracle attack [32] is an adaptive chosen ciphertext attack against PKCS#1 v1.5, the RSA padding standard used in SSL and TLS. It enables decryption of RSA ciphertexts if a server distinguishes between correctly and incorrectly padded RSA plaintexts, and was termed the "million-message attack" upon its introduction in 1998, after the number of decryption queries needed to deduce a plaintext. All widely used SSL/TLS servers include countermeasures against Bleichenbacher attacks.

Our first result shows that the SSLv2 protocol is fatally vulnerable to a form of Bleichenbacher attack that enables decryption of RSA ciphertexts. We develop a novel application of the attack that allows us to use a server that supports SSLv2 as an efficient padding oracle. This attack is a protocol-level flaw in SSLv2 that results in a feasible attack for 40-bit export cipher strengths, and in fact abuses the universally implemented countermeasures against Bleichenbacher attacks to obtain a decryption oracle.

We also discovered multiple implementation flaws in commonly deployed OpenSSL versions that allow an extremely efficient and much more dangerous instantiation of this attack.

Using SSLv2 to break TLS. Second, we present a novel *cross-protocol attack* that allows an attacker to break a passively collected RSA key exchange for any TLS server if the RSA keys are also used for SSLv2, possibly on a different server. We call this attack DROWN (*Decrypting RSA using Obsolete and Weakened eNcryption*).

In its *general* version, the attack exploits the protocol flaws in SSLv2, does not rely on any particular library implementation, and is feasible to carry out in practice by taking advantage of commonly supported export-grade ciphers. In order to decrypt one TLS session, the attacker must passively capture about 1,000 TLS sessions using RSA key exchange, make 40,000 SSLv2 connections to the victim server, and perform 2^{50} symmetric encryption operations. We successfully carried out this attack using an optimized GPU implementation and were able to decrypt a 2048-bit RSA ciphertext in less than 18 hours on a GPU cluster and less than 8 hours using Amazon EC2.

We found that 11.5 million HTTPS servers (33%) are vulnerable to this attack, because many HTTPS servers that do not directly support SSLv2 share RSA keys with other services that do. Of servers offering HTTPS with browser-trusted certificates, 22% are vulnerable.

We also present a *special* version of DROWN that exploits flaws in OpenSSL for a more efficient oracle. It requires roughly the same number of captured TLS sessions as the general attack, but only half as many connections to the victim server and no large computations. This attack can be completed on a single core on commodity hardware in less than a minute, and is limited primarily by how fast the server can complete handshakes. It is fast enough that an attacker can perform man-in-the-middle attacks on live TLS sessions before the handshake times out, and downgrade a modern TLS client to RSA key exchange with a server that prefers non-RSA cipher suites. Our Internet-wide scans suggest that 79% of HTTPS servers that are vulnerable to the general attack, or 26% of all HTTPS servers, are also vulnerable to real-time attacks exploiting these implementation flaws.

Our results highlight the risk that continued support for SSLv2 imposes on the security of much more recent TLS versions. This is an instance of a more general phenomenon of insufficient domain separation, where older, vulnerable security standards can open the door to attacks on newer versions. We conclude that phasing out outdated and insecure standards should become a priority for standards designers and practitioners.

Disclosure. DROWN was assigned CVE-2016-0800. We disclosed our attacks to OpenSSL and worked with them to coordinate further disclosures. The specific OpenSSL vulnerabilities we discovered have been designated CVE-2015-3197, CVE-2016-0703, and CVE-2016-0704. In response to our findings, OpenSSL has made it impossible to configure a TLS server in such a way that it is vulnerable to DROWN. Microsoft had already disabled SSLv2 for all supported versions of IIS. We also disclosed the attack to the NSS developers, who have disabled SSLv2 on the last NSS tool that supported it and have hastened efforts to entirely remove the protocol from their codebase. In response to our disclosure, Google will disable QUIC support for non-whitelisted servers and modify the QUIC standard. We also notified IBM, Cisco, Amazon, the German CERT-Bund, and the Israeli CERT.

4.2 Background

In the following, $a||b$ denotes concatenation of strings a and b. $a[i]$ references the i-th byte in a. (N,e) denotes an RSA public key, where N has byte-length ℓ_m ($|N| = \ell_m$) and e is the public exponent. The corresponding secret exponent is $d = 1/e \bmod \phi(N)$.

4.2.1 PKCS#1 v1.5 encryption padding

Our attacks rely on the structure of RSA PKCS#1 v1.5 padding. Although RSA PKCS#1 v2.0 implements OAEP, SSL/TLS still uses PKCS#1 v1.5. The PKCS#1 v1.5 encryption padding scheme [84] randomizes encryptions by prepending a random padding string PS to a message k (here, a symmetric session key) before RSA encryption:

1. The plaintext message is k, $\ell_k = |k|$. The encrypter generates a random byte string PS, where $|PS| \geq 8$, $|PS| = \ell_m - 3 - \ell_k$, and $\texttt{0x00} \notin \{PS[1], \ldots, PS[|PS|]\}$.

2. The encryption block is $m = 00||02||PS||00||k$.

3. The ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Then it checks whether the decrypted message m is correctly formatted as a PKCS#1 v1.5-encoded message. We say that the ciphertext c and the decrypted message bytes $m[1]||m[2]||...||m[\ell_m]$ are PKCS#1 v1.5 conformant if:

$$\begin{aligned} m[1]||m[2] &= \texttt{0x00}||\texttt{0x02} \\ \texttt{0x00} &\notin \{m[3], \ldots, m[10]\} \end{aligned}$$

If this condition holds, the decrypter searches for the first value $i > 10$ such that $m[i] = 0x00$. Then, it extracts $k = m[i+1]|| \ldots ||m[\ell_m]$. Otherwise, the ciphertext is rejected.

In SSLv3 and TLS, RSA PKCS#1 v1.5 is used to encapsulate the premaster secret exchanged during the handshake [42]. Thus, k is interpreted as the premaster secret. In SSLv2, RSA PKCS#1 v1.5 is used for encapsulation of an equivalent key denoted the `master_key`.

4.2.2 SSL and TLS

The first incarnation of the TLS protocol was the SSL (Secure Socket Layer) protocol, which was designed by Netscape in the 90s. The first two versions of SSL were immediately found to be vulnerable to trivial attacks [129, 133] which were fixed in SSLv3 [53]. Later versions of the standard were renamed TLS, and share a similar structure to SSLv3. The current version of the protocol is TLS 1.2; TLS 1.3 is currently under development.

An SSL/TLS protocol flow consists of two phases: handshake and application data exchange. In the first phase, the communicating parties agree on cryptographic algorithms and establish shared keys. In the second phase, these keys are used to protect the confidentiality and authenticity of the transmitted application data.

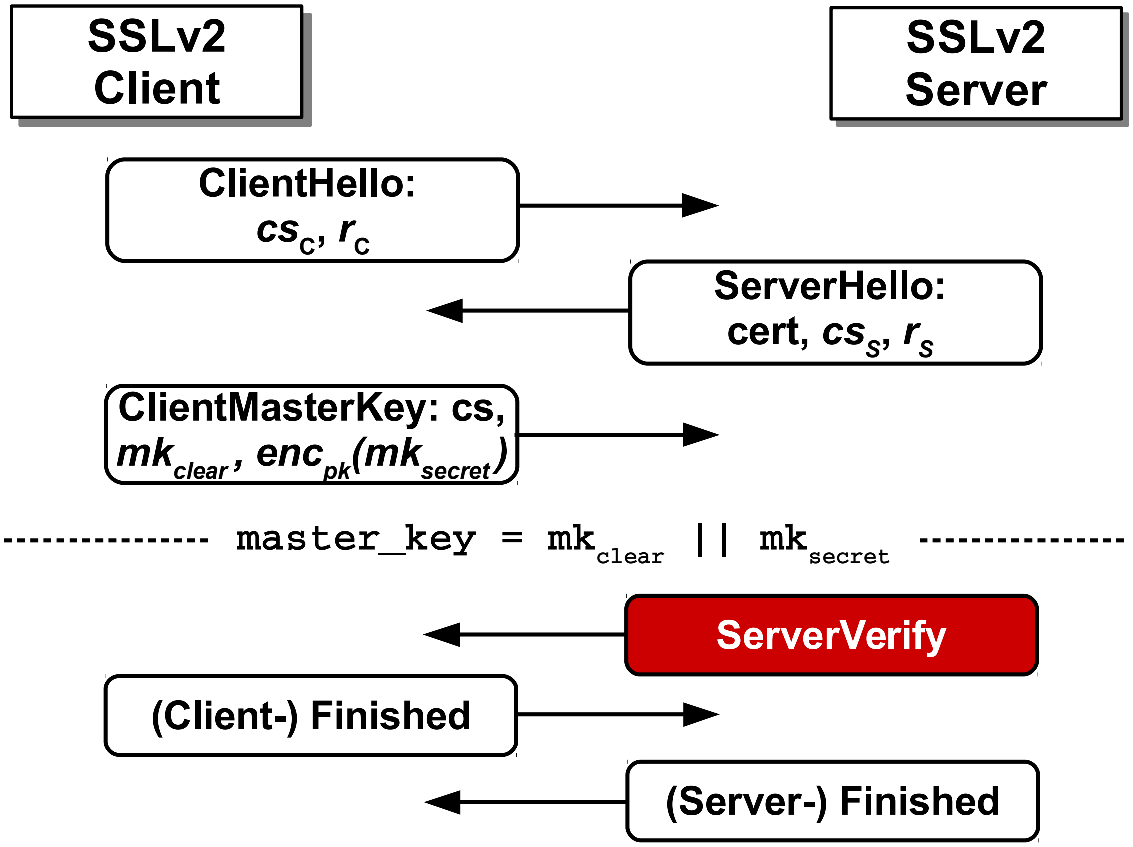
SSLv2 Client
SSLv2 Server
ClientHello: cs_C, r_C
ServerHello: cert, cs_S, r_S
ClientMasterKey: cs, mk_{clear}, $enc_{pk}(mk_{secret})$
master_key = mk_{clear} || mk_{secret}
ServerVerify
(Client-) Finished
(Server-) Finished

Figure 4.1: **SSLv2 handshake** — The server responds with a `ServerVerify` message directly after receiving an RSA-PKCS#1 v1.5 ciphertext contained in `ClientMasterKey`. This protocol feature enables the attack.

The handshake protocol was fundamentally redesigned in the SSLv3 version. This new handshake protocol was then used in later TLS versions up to TLS 1.2. In the following, we describe the RSA-based handshake protocols used in TLS and SSLv2, and highlight their differences.

The SSLv2 handshake protocol.

The SSLv2 protocol description [73] is less formally specified than modern RFCs. Figure 4.1 depicts an SSLv2 handshake.

A client initiates an SSLv2 handshake by sending a `ClientHello` message, which includes a list of cipher suites cs_c supported by the client and a client nonce r_c, termed `challenge`. The server responds with a `ServerHello` message, which contains a list of cipher suites cs_s supported by the server, the server certificate, and a server nonce r_s, termed `connection_ID`.

The client responds with a `ClientMasterKey` message, which specifies a cipher suite supported by both peers and key data used for constructing a `master_key`. In order to support *export* cipher suites with 40-bit security (e.g., SSL_RC2_128_CBC_EXPORT40_WITH_MD5), the key data is divided into two parts:

- mk_{clear}: A portion of the master_key sent in the ClientMasterKey message as plaintext (termed clear_key_data in the SSLv2 standard).
- mk_{secret}: A secret portion of the master_key, encrypted with RSA PKCS#1 v1.5 (termed secret_key_data).

The resulting master_key mk is constructed by concatenating these two keys: $mk = mk_{clear}||mk_{secret}$. For 40-bit export cipher suites, mk_{secret} is five bytes in length. For non-export cipher suites, the whole master_key is encrypted, and the length of mk_{clear} is zero.

The client and server can then compute session keys from the reconstructed master_key mk:

$$\texttt{server\_write\_key} = MD5(mk||\text{“0”}||r_c||r_s)$$
$$\texttt{client\_write\_key} = MD5(mk||\text{“1”}||r_c||r_s)$$

The server responds with a ServerVerify message consisting of the challenge r_c encrypted with the server_write_key. Both peers then exchange Finished messages in order to authenticate to each other.

Our attack exploits the fact that the server always decrypts an RSA-PKCS#1 v1.5 ciphertext, computes the server_write_key, and *immediately* responds with a ServerVerify message. The SSLv2 standard implies this message ordering, but does not make it explicit. However, we observed this behavior in every implementation we examined. Our attack also takes advantage of the fact that the encrypted mk_{secret} portion of the master_key can vary in length, and is only five bytes for export ciphers.

The TLS handshake protocol. In TLS [42] or SSLv3, the client initiates the handshake with a ClientHello, which contains a client random r_c and a list of supported cipher suites. The server chooses one of the cipher suites and responds with three messages, ServerHello, Certificate, and ServerHelloDone. These messages include the server's choice of cipher suite, server nonce r_s, and a server certificate with an RSA public key. The client then uses the public key to encrypt a newly generated 48-byte premaster secret *pms* and sends it to the server in a ClientKeyExchange message. The client and server then derive encryption and MAC keys from the premaster secret and the client and server random nonces. The details of this derivation are not important to our attack. The client then sends ChangeCipherSpec and Finished messages. The Finished message authenticates all previous handshake messages using the derived keys. The server responds with its own ChangeCipherSpec and Finished messages.

The two main details relevant to our attacks are:

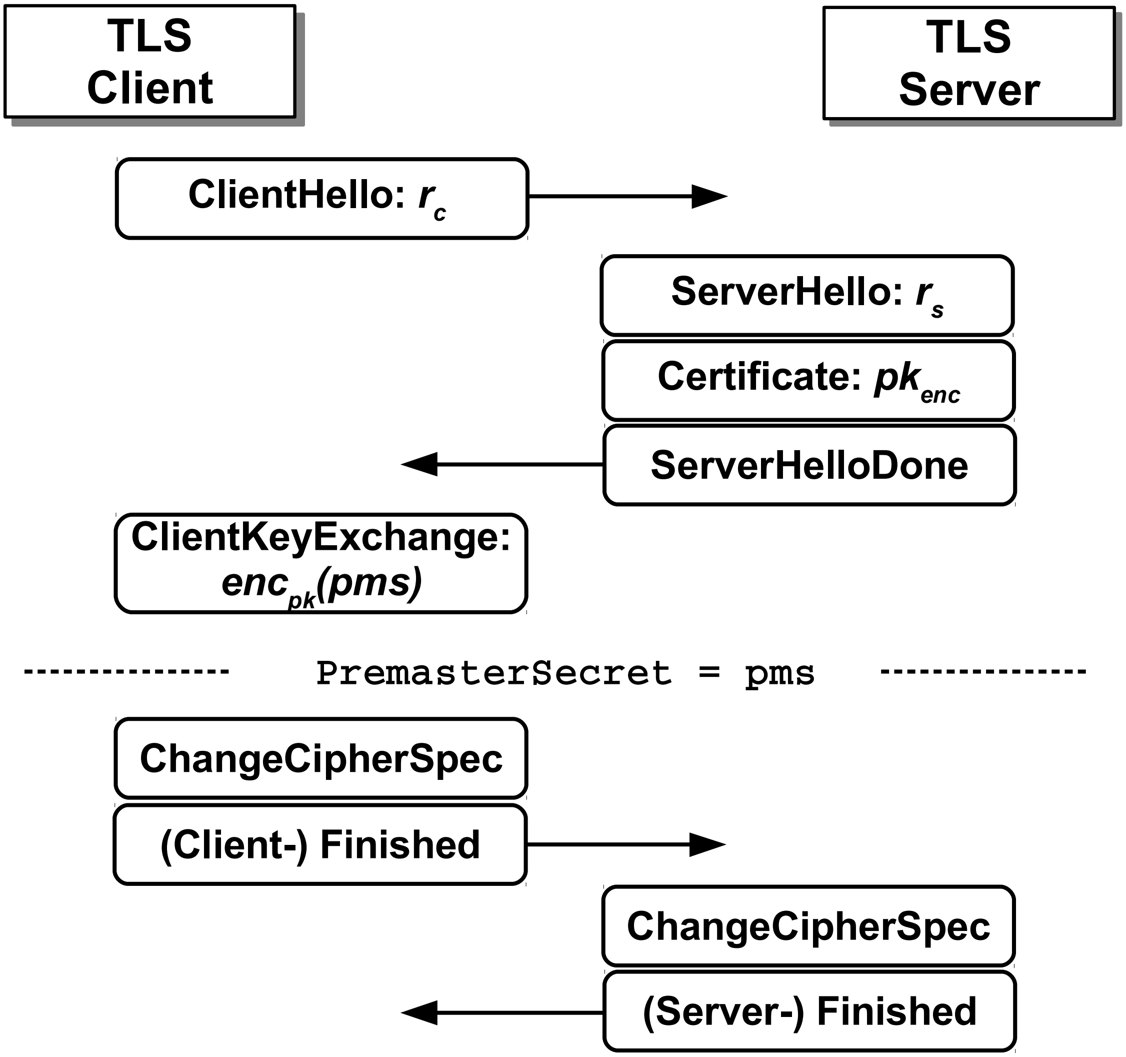

Figure 4.2: **TLS-RSA handshake** — After receiving an encrypted premaster secret, the server waits for an authenticated `ClientFinished` message.

- The premaster secret is always 48 bytes long, independent of the chosen cipher suite. This is also true for export cipher suites.
- After receiving the `ClientKeyExchange` message, the server waits for the `ClientFinished` message, in order to authenticate the client.

4.2.2.1 Real-world protocol support

TLSv1.0 is the most commonly supported protocol version, according to several surveys. The SSL Labs SSL Pulse survey [119] reports that 98.6% of about 140,000 popular TLS/SSL-enabled web sites supported TLSv1.0 in January 2016. 72.0% supported TLSv1.2. Support for SSLv2 was at 9.3%, and SSLv3 was at 29%. Mayer et al. [102] performed Internet-wide surveys of SMTP, IMAP, and POP3 between April and August 2015, and found that support for SSLv2 support was as high as 41.7% of servers for SMTP on port 25 and as low as 3.7% of IMAP servers on port 143. Support for TLSv1.0 was nearly universal on these ports, varying from 91.6% on port 25 to 98.9% on port 143.

Bowen [34] collected 213 million SSL/TLS client hellos and user agent strings from connections to popular sites, of which 183,000 (0.09%) client hellos supported SSLv2. All of these client hellos also supported at least TLSv1.0.

Holz et al. [75] performed passive monitoring to collect information about 16 million SSL/TLS connections during one week in July-August 2015. They did not report any numbers for SSLv2, and stated in personal communication that they did not observe any SSLv2 connections in their dataset.

4.2.3 Bleichenbacher's attack

Bleichenbacher's attack is a padding oracle attack—it exploits the fact that RSA ciphertexts should decrypt to PKCS#1 v1.5-compliant plaintexts. If an implementation receives an RSA ciphertext that decrypts to an invalid PKCS#1 v1.5 plaintext, it might naturally leak this information via an error message, by closing the connection, or by taking longer to process the error condition. This behavior can leak information about the plaintext that can be modeled as a cryptographic *oracle* for the decryption process. Bleichenbacher [32] demonstrated how such an oracle could be exploited to decrypt RSA ciphertexts.

Algorithm. In the simplest attack scenario, the attacker has a valid PKCS#1 v1.5 ciphertext c_0 that they wish to decrypt to discover the message m_0. They have no access to the private RSA key, but instead have access to an oracle $\mathcal{O}$ that will decrypt a ciphertext c and inform the attacker whether the most significant two bytes match the required value for a correct

PKCS#1 v1.5 padding:

$$\mathscr{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with `1`, the attacker knows that $2B \leq m \leq 3B - 1$, where $B = 2^{8(\ell_m - 2)}$. The attacker can take advantage of RSA malleability to generate new candidate ciphertexts for any s:

$$c = (c_0 \cdot s^e) \bmod N = (m_0 \cdot s)^e \bmod N$$

The attacker queries the oracle with c. If the oracle responds with 0, the attacker increments s and repeats the previous step. Otherwise, the attacker learns that for some r, $2B \leq m_0 s - rN < 3B$. This allows the attacker to reduce the range of possible solutions to:

$$\frac{2B + rN}{s} \leq m_0 < \frac{3B + rN}{s}$$

The attacker proceeds by refining guesses for s and r values and successively decreasing the size of the interval containing m_0. At some point the interval will contain a single valid value, m_0. Bleichenbacher's original paper describes this process in further detail [32].

Countermeasures. In order to protect against this attack, the decrypter must not leak information about the PKCS#1 v1.5 validity of the ciphertext. The ciphertext does not decrypt to a valid message, so the decrypter generates a fake plaintext and continues the protocol with this decoy. The attacker should not be able to distinguish the resulting computation from a correctly decrypted ciphertext.

In the case of SSL/TLS, the server generates a random premaster secret to continue the handshake if the decrypted ciphertext is invalid. The client will not possess the session key to send a valid `ClientFinished` message and the connection will terminate.

4.3 Breaking TLS with SSLv2

In this section, we describe our cross-protocol DROWN attack that uses an SSLv2 server as an oracle to efficiently decrypt TLS connections. The attacker learns the session key for targeted TLS connections but does not learn the server's private RSA key. We first describe our techniques using a generic SSLv2 oracle. In Section 4.4.1, we show how a protocol flaw in SSLv2 can be used to construct such an oracle, and describe our general DROWN attack. In Section 4.5, we show how an implementation flaw in common versions of OpenSSL leads to a more powerful oracle and describe our efficient special DROWN attack.

We consider a server accepting TLS connections from clients. The connections are established using a secure, state-of-the-art TLS version (1.0–1.2) and a `TLS_RSA` cipher suite with a private key unknown to the attacker.

The same RSA public key as the TLS connections is also used for SSLv2. For simplicity, we will refer to the servers accepting TLS and SSLv2 connections as the same entity.

Our attacker is able to passively eavesdrop on traffic between the client and server and record RSA-based TLS traffic. The attacker may or may not be also required to perform active man-in-the-middle interference, as explained below.

The attacker can expect to decrypt one out of 1,000 intercepted TLS connections in our attack for typical parameters. This is a devastating threat in many scenarios. For example, a decrypted TLS connection might reveal a client's HTTP cookie or plaintext password, and an attacker would only need to successfully decrypt a single ciphertext to compromise the client's account. In order to collect 1,000 TLS connections, the attacker might simply wait patiently until sufficiently many connections are recorded. A less patient attacker might use man-in-the-middle interference, as in the BEAST attack [44].

4.3.1 A generic SSLv2 oracle

Our attacks make use of an oracle that can be queried on a ciphertext and leaks information about the decrypted plaintext; this abstractly models the information gained from an SSLv2 server's behavior. Our SSLv2 oracles reveal many bytes of plaintext, enabling an efficient attack.

Our cryptographic oracle $\mathcal{O}$ has the following functionality: $\mathcal{O}$ decrypts an RSA ciphertext c and responds with ciphertext validity based on the decrypted message m. The ciphertext is valid only if m starts with `0x00 02` followed by non-null padding bytes, a delimiter byte `0x00`, and a `master_key` mk_{secret} of correct byte length ℓ_k. We call such a ciphertext *SSLv2 conformant*.

All of the SSLv2 padding oracles we instantiate give the attacker similar information about a PKCS#1 v1.5 conformant SSLv2 ciphertext:

$$\mathcal{O}(c) = \begin{cases} mk_{secret} & \text{if } c^d \bmod N = 00||02||PS||00||mk_{secret} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the oracle $\mathcal{O}(c)$ will return the decrypted message mk_{secret} if it is queried on a PKCS#1 v1.5 conformant SSLv2 ciphertext c corresponding to a correctly PKCS#1 v1.5 padded encryption of mk_{secret}. The attacker then learns $\ell_k + 3$ bytes of $m = c^d \bmod N$: the first two bytes are $00||02$, and the last $\ell_k + 1$ bytes are $00||mk_{secret}$. The length ℓ_k of mk_{secret} varies based on the cipher suite used to instantiate the oracle. For export-grade cipher suites such as SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5, k will be 5 bytes, so the attacker learns 8 bytes of m.

4.3.2 DROWN attack template

Our attacker will use an SSLv2 oracle $\mathcal{O}$ to decrypt a TLS `ClientKeyExchange`. The behavior of $\mathcal{O}$ poses two problems for the attacker. First, a TLS key exchange ciphertext decrypts to a 48-byte premaster secret. But since no SSLv2 cipher suites have 48-byte key strengths, this means that a valid TLS ciphertext is invalid to our oracle $\mathcal{O}$. In order to apply Bleichenbacher's attack, the attacker must transform the TLS ciphertext into a valid SSLv2 key exchange message. Second, $\mathcal{O}$ is very restrictive, since it strictly checks the length of the unpadded message. According to Bardou et al. [23], Bleichenbacher's attack would require 12 million queries to such an oracle.[1]

Our attacker overcomes these problems by following this generic attack flow:

0. The attacker collects many encrypted TLS RSA key exchange messages.

1. The attacker converts one of the intercepted TLS ciphertexts containing a 48-byte premaster secret to an RSA PKCS#1 v1.5 encoded ciphertext valid to the SSLv2 oracle $\mathcal{O}$.

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext, they can continue with a modified version of Bleichenbacher's attack, and decrypt the message after many more oracle queries.

3. The attacker then transforms the decrypted plaintext back into the original plaintext, which is one of the collected TLS handshakes.

We describe the algorithmic improvements we use to make each of these steps efficient below.

4.3.2.1 Finding an SSLv2 conformant ciphertext

The first step for the attacker is to transform the original TLS `ClientKeyExchange` message c_0 from a TLS conformant ciphertext into an SSLv2 conformant ciphertext.

For this task, we rely on the concept of *trimmers*, which were introduced by Bardou et al. [23]. Assume that the message $m_0 = c_0{}^d \bmod N$ is divisible by a small number t. In that case, $m_0 \cdot t^{-1} \bmod N$ simply equals the natural number m_0/t. If we choose $u \approx t$, and multiply the original message by $u \cdot t^{-1}$, the resulting number will lie near the original message: $m_0 \approx m_0/t \cdot u$.

[1] See Table 1 in [23]. The oracle is denoted with the term `FFF`.

This method gives a good chance of generating a new SSLv2 conformant message. Let c_0 be an intercepted TLS conformant RSA ciphertext, and let $m_0 = c_0^d \bmod N$ be the plaintext. We select a multiplier $s = u/t \bmod N = ut^{-1} \bmod N$ where u and t are coprime, compute the value $c_1 = c_0 s^e \bmod N$, and query $\mathcal{O}(c_1)$. We will receive a response if $m_1 = m_0 \cdot u/t$ is SSLv2 conformant.

As an example, let us assume a 2048-bit RSA ciphertext with $\ell_k = 5$, and consider the fraction $u = 7, t = 8$. The probability that $c_0 \cdot u/t$ will be SSLv2 conformant is 1/7,774, so we expect to make 7,774 oracle queries before obtaining a positive response from $\mathcal{O}$. Appendix 4.11.1 gives more details on computing these probabilities.

4.3.2.2 Shifting known plaintext bytes

Once we have obtained an SSLv2 conformant ciphertext c_1, the oracle has also revealed the $\ell_k + 1$ least significant bytes (mk_{secret} together with the delimiter byte 0x00) and two most significant 0x00 02 bytes of the SSLv2 conformant message m_1. We would like to *rotate* these known bytes around to the right, so that we have a large block of contiguous known most significant bytes of plaintext. In this section, we show that this can be accomplished by multiplying by some shift $2^{-r} \bmod N$. In other words, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot 2^{-r} \bmod N$ and we know several most significant bytes of m_2.

Let $R = 2^{8(k+1)}$ and $B = 2^{8(\ell_m - 2)}$. Abusing notation slightly, let the integer $m_1 = 2 \cdot B + PS \cdot R + mk_{secret}$ be the plaintext satisfying $m_1^e = c_1 \bmod N$. At this stage, the ℓ_k-byte integer mk_{secret} is known and the $\ell_m - \ell_k - 3$-byte integer PS is not.

Let $\tilde{m_1} = 2 \cdot B + mk_{secret}$ be the known components of m_1, so $m_1 = \tilde{m_1} + PS \cdot R$. We can use this to compute a new plaintext for which we know many most significant bytes. Consider the value:

$$m_1 \cdot R^{-1} \bmod N = \tilde{m_1} \cdot R^{-1} + PS \bmod N.$$

The value of PS is unknown and consists of $\ell_m - \ell_k - 3$ bytes. This means that the known value $\tilde{m_1} \cdot R^{-1}$ shares most of its $\ell_k + 3$ most significant bytes with $m_1 \cdot R^{-1}$.

Furthermore, we can iterate this process by finding a new multiplier s such that $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ is also SSLv2 conformant. A randomly chosen $s < 2^{30}$ will work with probability $2^{-25.4}$. We can take use the bytes we have already learned about m_1 to efficiently compute such an s with only 678 oracle queries in expectation for a 2048-bit RSA modulus. Appendix 4.11.3 gives more details.

4.3.2.3 Adapted Bleichenbacher iteration

It is feasible for all of our oracles to use the previous technique to entirely recover a plaintext message. However, for our SSLv2 protocol oracle it is cheaper after a few iterations to continue using Bleichenbacher's original attack. We can apply the original algorithm proposed by Bleichenbacher as described in Section 4.2.3.

Each step obtains a message that starts with the required `0x00 02` bytes after two queries in expectation. Since we know the value of the $\ell_k + 1$ least significant bytes after multiplying by any integer, we can query the oracle only on multipliers that cause the $(\ell_k + 1)$st least significant byte to be zero. However, we cannot ensure that the padding string is entirely nonzero; for a 2048-bit modulus this will hold with probability 0.37.

For a 2048-bit modulus, the total expected number of queries when using this technique to fully decrypt the plaintext is $2048 * 2/0.37 \approx 11,000$.

4.4 General DROWN

In this section, we describe how to use any correct SSLv2 implementation accepting export-grade cipher suites as a padding oracle. We then show how to adapt the techniques described in Section 4.3.2 to decrypt TLS RSA ciphertexts.

4.4.1 The SSLv2 export padding oracle

SSLv2 is vulnerable to a direct message side channel vulnerability exposing a Bleichenbacher oracle to the attacker. The vulnerability follows from three properties of SSLv2. First, the server immediately responds with a `ServerVerify` message after receiving the `ClientMasterKey` message, which includes the RSA ciphertext, without waiting for the `ClientFinished` message that proves the client knows the RSA plaintext. Second, when choosing 40-bit export RC2 or RC4 as the symmetric cipher, only 5 bytes of the `master_key` (mk_{secret}) are sent encrypted using RSA, and the remaining 11 bytes are sent in cleartext. Third, a server implementation that correctly implements the anti-Bleichenbacher countermeasure and receives an RSA key exchange message with invalid padding will generate a random premaster secret and carry out the rest of the TLS handshake using this randomly generated key material.

This allows an attacker to deduce the validity of RSA ciphertexts in the following manner:

1. The attacker sends a `ClientMasterKey` message, which contains an RSA ciphertext c_0 and any choice of 11 clear key bytes for mk_{clear}. The server responds with a `ServerVerify` message, which contains the `challenge` encrypted using the `server_write_key`.

2. The attacker performs an *exhaustive search* over the possible values of the 5 bytes of the master_key mk_{secret}, computes the corresponding server_write_key, and checks whether the ServerVerify message decrypts to challenge. One value should pass this check; call it mk_0. Recall that if the RSA plaintext was valid, mk_0 is the unpadded data in the RSA plaintext c_0^d. Otherwise, mk_0 is a randomly generated sequence of 5 bytes.

3. The attacker re-connects to the server with the same RSA ciphertext c_0. The server responds with another ServerVerify message that contains the current challenge encrypted using the current server_write_key. If the decrypted RSA ciphertext was valid, the attacker can use mk_0 to decrypt a correct challenge value from the ServerVerify message. Otherwise, if the ServerVerify message does not decrypt to challenge, the RSA ciphertext was invalid, and mk_0 must have been random.

Thus we can instantiate an oracle $\mathcal{O}_{\text{SSLv2-export}}$ using the procedure above; each oracle query requires two server connections and 2^{40} decryption attempts in the simplest case. For each oracle call $\mathcal{O}_{\text{SSLv2-export}}(c)$, the attacker learns whether c is valid, and if so, learns the two most significant bytes 0x00 02, the sixth least significant 0x00 delimiter byte, and the value of the 5 least significant bytes of the plaintext m.

4.4.2 OpenSSL special DROWN oracle

We discovered a vulnerability present in OpenSSL versions prior to March 4, 2015 that allows a client to improperly provide cleartext key bytes for non-export ciphers. Affected servers will substitute these bytes for bytes from the encrypted key. This allows a client to successively learn a byte at a time of an encrypted key by brute forcing only 256 possibilities for each query. For a non-export 128-bit cipher suite such as SSL_RC4_WITH_MD5, the attacker learns 19 bytes of the decrypted message. We describe this vulnerability in more detail in Appendix 4.5.1. A client can then construct a Bleichenbacher oracle from this behavior by validating the ServerVerify message against the candidate key provided in the clear_key_data, resulting in no brute-force computation.

4.4.3 TLS decryption attack

In this section, we describe how the oracle described in Section 4.4.1 can be used to carry out a feasible attack to decrypt passively collected TLS ciphertexts.

As described in Section 4.3, we consider a server that accepts TLS connections from clients using an RSA public key that is exposed via SSLv2, and an attacker who is able to passively observe these connections.

We also assume the server supports export cipher suites for SSLv2. This can happen for two reasons. First, the same server operators that fail to follow best practices in disabling SSLv2 [129] may also fail to follow best practices by supporting export cipher suites. Alternatively, the server might be running a version of OpenSSL prior to January 2016, in which case it is vulnerable to the OpenSSL cipher suite selection bug described in Section 4.7, and an attacker may negotiate a cipher suite of his choice independent of the server configuration.

The attacker needs access to computing power sufficient to perform a 2^{50} time attack, mostly brute forcing symmetric key encryption. After our optimizations, this can be done with a one-time investment of a few thousand dollars of GPUs, or in a few hours for a few hundred dollars in the cloud. Our cost estimates are described in Section 4.4.4.

4.4.3.1 Constructing the attack

The attacker can exploit the SSLv2 vulnerability following the generic attack outline described in Section 4.3.2, consisting of several distinct phases:

0. The attacker passively collects 1,000 TLS handshakes from connections using RSA key exchange.

1. They then attempt to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing five-byte messages using the fractional trimmers described in Section 4.3.2.1, and querying $\mathcal{O}_{\text{SSLv2-export}}$. The attacker sends the modified ciphertexts to the server using fresh SSLv2 connections with weak symmetric ciphers and uses the `ServerVerify` messages to deduce ciphertext validity as described in the previous section. For each queried RSA ciphertext, the attacker must perform a brute force attack on the weak symmetric cipher. The attacker expects to obtain a valid SSLv2 ciphertext after roughly 10,000 oracle queries, or 20,000 connections to the server.

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext $c_1 = m_1^e$, they use the shifting technique explained in Section 4.3.2.2 to find an integer s_1 such that $m_2 = m_1 \cdot 2^{-40} \cdot s_1$ is also SSLv2 conformant. Appendix 4.11.4 contains more details on this step.

3. The attacker then applies the shifting technique again to find another integer s_2 such that $m_3 = m_2 \cdot 2^{-40} \cdot s_2$ is also SSLv2 conformant.

4. They then search for yet another integer s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant.

Optimizing for	**Cipher-texts**	$\|F\|$	**SSLv2 connections**	**Offline work**
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

Table 4.1: **2048-bit Bleichenbacher attack complexity** — The cost to decrypt one ciphertext can be adjusted by choosing the set of fractions F the attacker applies to each of the passively collected ciphertexts in the first step of the attack. This choice affects several parameters: the number of these collected ciphertexts, the number of connections the attacker makes to the SSLv2 server, and the number of offline decryption operations.

5. Finally, the attacker can continue with our adapted Bleichenbacher iteration technique described in Section 4.3.2.3, and decrypts the message after an expected 10,000 additional oracle queries, or 20,000 connections to the server.

6. The attacker can then transform the decrypted plaintext back into the original plaintext, which is one of the 1,000 intercepted TLS handshakes.

The rationale behind the different phases. Bleichenbacher's original algorithm requires a conformant message m_0, and a multiplier s_1 such that $m_1 = m_0 \cdot s_1$ is also conformant. Naïvely, it would appear we can apply the same algorithm here, after completing Phase 1. However, the original algorithm expects s_1 to be of size about 2^{24}. This is not the case when we use fractions for s_1, as the integer $s_1 = ut^{-1} \bmod N$ will be the same size as N.

Therefore, our approach is to find a conformant message for which we know the 5 most significant bytes; this will happen after multiple rotations and this message will be m_3. After finding such a message, finding s_3 such that $m_4 = m_3 \cdot s_3$ is also conformant becomes trivial. From there, we can finally apply the adapted Bleichenbacher iteration technique as described in Appendix 4.11.5.

4.4.3.2 Attack performance

The attacker wishes to minimize three major costs in the attack: the number of recorded ciphertexts from the victim client, the number of connections to the victim server, and the number of symmetric keys to be brute forced. The requirements for each of these elements are governed by the set of fractions to be multiplied with each RSA ciphertext in the first phase, as described in Section 4.3.2.1.

Key size	Phase 1	Phases 2–5	Total queries	Offline work
1024	4,129	4,132	8,261	$2^{50.01}$
2048	6,919	12,468	19,387	$2^{50.76}$
4096	18,286	62,185	80,471	$2^{52.16}$

Table 4.2: **Oracle queries required by our attack** — In Phase 1, the attacker queries the oracle until an SSLv2 conformant ciphertext is found. In Phases 2–5, the attacker decrypts this ciphertext using leaked plaintext. These numbers minimize total queries. In our attack, an oracle query represents two server connections.

Table 4.1 highlights a few choices for F and the resulting performance metrics for 2048-bit RSA keys. Appendix 4.11 provides more details on the derivation of these numbers and other optimization choices. Table 4.2 gives the expected number of Bleichenbacher queries for different RSA key sizes, when minimizing total oracle queries.

4.4.4 Implementing general DROWN with GPUs

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key, so we developed a highly optimized GPU implementation of this brute force attack. Our first naïve GPU implementation performed around 26MH/s, where MH denotes the time required for testing one million possible values of mk_{secret}. Our optimized implementation runs at a final speed of 515MH/s, a speedup factor of 19.8.

We obtained our improvements through a number of optimizations. For example, our original implementation ran into a communication bottleneck in the PCI-E bus in transmitting candidate keys from CPU to GPU, so we removed this bottleneck by generating key candidates on the GPU itself. We optimized memory management, including storing candidate keys and the RC2 permutation table in constant memory, which is almost as fast as a register, instead of slow global memory.

We experimentally evaluated our optimized implementation on a local cluster and in the cloud. We used it to execute a full attack of $2^{49.6}$ tested keys on each platform. The required number of keys to test during the attack is a random variable, distributed geometrically, with an expectation that ranges between $2^{49.6}$ and $2^{52.5}$ depending on the choice of optimization parameters. We treat a full attack as requiring $2^{49.6}$ tested keys overall.

Hashcat. Hashcat [71] is an open source optimized password-recovery tool. The Hashcat developers allowed us to use their GPU servers for our attack evaluation. The servers contain a total of 40 GPUs: 32 Nvidia GTX 980 cards, and 8 AMD R9 290X cards. The value of this equipment is roughly \$18,040. Our full attack took less than 18 hours to complete on the Hashcat servers, with the longest single instance taking 17h9m.

Amazon EC2. We also ran our optimized GPU code on the Amazon Elastic Compute Cloud (EC2) service. We used a cluster composed of 200 variable-price "spot" instances: 150 `g2.2xlarge` instances, each containing one high-performance NVIDIA GPU with 1,536 CUDA cores and 50 `g2.8xlarge` instances, each containing four of these GPUs. When we ran our experiments in January 2016, the average spot rates we paid were $0.09/hr and $0.83/hr respectively. Our full attack finished in under 8 hours including startup and shutdown for a cost of $440.

4.4.5 OpenSSL SSLv2 cipher suite selection bug

General DROWN is a protocol flaw, but the population of vulnerable hosts is increased due to a bug in OpenSSL that causes many servers to erroneously support SSLv2 and export ciphers even when configured not to. The OpenSSL team intended to disable SSLv2 by default in 2010, with a change that removed all SSLv2 cipher suites from the default list of ciphers offered by the server [144]. However, the code for the protocol itself was not removed in standard builds and SSLv2 itself remained enabled. We discovered a bug in OpenSSL's SSLv2 cipher suite negotiation logic that allows clients to select SSLv2 cipher suites even when they are not explicitly offered by the server. We notified the OpenSSL team of this vulnerability, which was assigned CVE-2015-3197. The problem was fixed in OpenSSL releases 1.0.2f and 1.0.1r [144].

4.5 Special DROWN

We discovered multiple vulnerabilities in recent (but not current) versions of the OpenSSL SSLv2 handshake code that create even more powerful Bleichenbacher oracles, and drastically reduce the amount of computation required to implement our attacks. The vulnerabilities, designated CVE-2016-0703 and CVE-2016-0704, were present in the OpenSSL codebase from at least the start of the repository, in 1998, until they were unknowingly fixed on March 4, 2015 by a patch [86] designed to correct an unrelated problem [40]. By adapting DROWN to exploit this special case, we can significantly cut both the number of connections and the computational work required.

4.5.1 The OpenSSL "extra clear" oracle

Prior to the fix, OpenSSL servers improperly allowed the `ClientMasterKey` message to contain `clear_key_data` bytes for *non-export* ciphers. When such bytes are present, the server substitutes them for bytes from the encrypted key. For example, consider the case that the client chooses a 128-bit cipher and sends a 16-byte encrypted key $k[1], k[2], \ldots, k[16]$ but, contrary to the protocol specification, includes 4 null bytes of clear_key_data. Vulnerable OpenSSL versions will construct the following master_key:

[00 00 00 00 $k[1]$ $k[2]$ $k[3]$ $k[4]$ ... $k[9]$ $k[10]$ $k[11]$ $k[12]$]

This enables a straightforward key recovery attack against such versions. An attacker that has intercepted an SSLv2 connection takes the RSA ciphertext of the encrypted key and replays it in non-export handshakes to the server with varying lengths of clear_key_data. For a 16-byte encrypted key, the attacker starts with 15 bytes of clear key, causing the server to use the master_key:

[00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 $k[1]$]

The attacker can brute force the first byte of the encrypted key by finding the matching ServerVerify message among 256 possibilities. Knowing $k[1]$, the attacker makes another connection with the same RSA ciphertext but 14 bytes of clear key, resulting in the master_key:

[00 00 00 00 00 00 00 00 00 00 00 00 00 00 $k[1]$ $k[2]$]

The attacker can now easily brute force $k[2]$. With only 15 probe connections and an expected $15 \cdot 128 = 1,920$ trial encryptions, the attacker learns the entire master_key for the recorded session.

As this oracle is obtained by improperly sending unexpected clear-key bytes, we call it the Extra Clear oracle.

This session key-recovery attack can be directly converted to a Bleichenbacher oracle. Given a candidate ciphertext and symmetric key length ℓ_k, the attacker sends the ciphertext with ℓ_k known bytes of clear_key_data. The oracle decision is simple:

- If the ciphertext is valid, the ServerVerify message will reflect a master_key consisting of those ℓ_k known bytes.
- If the ciphertext is invalid, the master_key will be replaced with ℓ_k random bytes (by following the countermeasure against the Bleichenbacher attack), resulting in a different ServerVerify message.

This oracle decision requires one connection to the server and one ServerVerify computation. After the attacker has found a valid ciphertext corresponding to a ℓ_k-byte encrypted key, they recover the ℓ_k plaintext bytes by repeating the key recovery attack from above. Thus our oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}(c)$ requires one connection to determine whether c is valid. After ℓ_k connections, the attacker additionally learns the ℓ_k least significant bytes of m. We model this as a single oracle call, but the number of server connections will vary depending on the response.

4.5.2 MITM attack against TLS

Special DROWN is fast enough that it can decrypt a TLS premaster secret *online*, during a connection handshake. A man-in-the-middle attacker can use it to compromise connections between modern browsers and TLS servers—even those configured to prefer non-RSA cipher suites.

The MITM attacker impersonates the server and sends a `ServerHello` message that selects a cipher suite with RSA as the key-exchange method. Then, the attacker uses special DROWN to decrypt the premaster secret. The main difficulty is completing the decryption and producing a valid `ServerFinished` message before the client's connection times out. Most browsers will allow the handshake to last up to one minute [17].

The attack requires targeting an average of 100 connections, only one of which will be attacked, probabilistically. The simplest way for the attacker to facilitate this is to use JavaScript to cause the client to connect repeatedly to the victim server, as described in Section 4.3. Each connection is tested against the oracle with only small number of fractions, and the attacker can discern immediately when he receives a positive response from the oracle.

Note that since the decryption must be completed online, the Leaky Export oracle cannot be used, and the attack uses only the Extra Clear oracle.

4.5.2.1 Constructing the attack

We will use `SSL_DES_192_EDE3_CBC_WITH_MD5` as the cipher suite, allowing the attacker to recover 24 bytes of key at a time. The attack works as follows:

0. The attacker causes the victim client to connect repeatedly to the victim server, with at least 100 connections.

1. The attacker uses the fractional trimmers as described in Section 4.3.2.1 to convert one of the TLS ciphertexts into an SSLv2 conformant ciphertext c_0.

2. Once the attacker has obtained a valid SSLv2 ciphertext c_1, they repeatedly use the shifting technique described in Section 4.3.2.2 to rotate the message by 25 bytes each iteration, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext.

3. The attacker then transforms the decrypted SSLv2 plaintext into the decrypted TLS plaintext.

Using 100 fractional trimmers, this more efficient oracle attack allows the attacker to recover one in 100 TLS session keys using only about 27,000 connections to the server, as described in Appendix 4.11.6. The computation cost is so low that we can complete the full attack on a single workstation in under one minute.

4.5.3 The OpenSSL "leaky export" oracle

In addition to the extra clear implementation bug, the same set of OpenSSL versions also contain a separate bug, where they do not follow the correct algorithm for their implementation of the Bleichenbacher countermeasure. We now describe this faulty implementation:

- The SSLv2 `ClientKeyExchange` message contains the mk_{clear} bytes immediately before the ciphertext c. Let p be the buffer starting at the first mk_{clear} byte.

- Decrypt c in place. If the decryption operation succeeds, and c decrypted to a plaintext of a correct padded length, p now contains the 11 mk_{clear} bytes followed by the 5 mk_{secret} bytes.

- If c decrypted to an unpadded plaintext k of incorrect length, the decryption operation overwrites the first $j = min(|k|, 5)$ bytes of c with the first j bytes of k.

- If c is not SSLv2 conformant and the decryption operation failed, randomize the first five bytes of p, which are the first five bytes of mk_{clear}.

This behavior allows the attacker to distinguish between these three cases. Suppose the attacker sends 11 null bytes as mk_{clear}. Then these are the possible cases:

1. c decrypts to a correctly padded plaintext k of the expected length, 5 bytes. Then the following `master_key` will be constructed:

   [00 00 00 00 00 00 00 00 00 00 00 $k[1]$ $k[2]$ $k[3]$ $k[4]$ $k[5]$]

2. c decrypts to a correctly padded plaintext k of a wrong length. Let r be the five random bytes the server generated. The yielded `master_key` will be:

   [$r[1]$ $r[2]$ $r[3]$ $r[4]$ $r[5]$ 00 00 00 00 00 00 $k[1]$ $k[2]$ $k[3]$ $k[4]$ $k[5]$]

   when $|k| \geq 5$. If $|k| < 5$, the server substitutes the first $|k|$ bytes of c with the first $|k|$ bytes of k. Using $|k| = 3$ as an example, the `master_key` will be:

   [$r[1]$ $r[2]$ $r[3]$ $r[4]$ $r[5]$ 00 00 00 00 00 00 $k[1]$ $k[2]$ $k[3]$ $c[4]$ $c[5]$]

3. c is not SSLv2 conformant, and hence the decryption operation failed. The resulting `master_key` will be:

   [$r[1]$ $r[2]$ $r[3]$ $r[4]$ $r[5]$ 00 00 00 00 00 00 $c[1]$ $c[2]$ $c[3]$ $c[4]$ $c[5]$]

The attacker detects case (3) by performing an exhaustive search over the 2^{40} possibilities for r, and checking whether any of the resulting values for the `master_key` correctly decrypts the observed `ServerVerify` message. If no r value satisfies this property, then c^d starts with bytes `0x00 02`. The attacker then distinguishes between cases (1) and (2) by performing an exhaustive search over the five bytes of k, and checking whether any of the resulting values for mk correctly decrypts the observed `ServerVerify` message.

As this oracle leaks information when using export ciphers, we have named it the Leaky Export oracle.

In conclusion, $\mathscr{O}_{\textsf{SSLv2-export-leaky}}$ allows an attacker to obtain a valid oracle response for all ciphertexts which decrypt to a correctly-padded plaintext of *any* length. This is in contrary to the previous oracles $\mathscr{O}_{\textsf{SSLv2-extra-clear}}$ and $\mathscr{O}_{\textsf{SSLv2-export}}$, which required the plaintext to be of a specific length. Each oracle query to $\mathscr{O}_{\textsf{SSLv2-export-leaky}}$ requires one connection to the server and 2^{41} offline work.

Combining the two oracles.

The attacker can use the Extra Clear and Leaky Export oracles together in order to reduce the number of queries required for the TLS decryption attack. They first test a TLS conformant ciphertext for divisors using the Leaky Export oracle, then use fractions dividing the plaintext with both oracles. Once the attacker has obtained a valid SSLv2 ciphertext c_1, they repeatedly use the shifting technique described in Section 4.3.2.2 to rotate the message by 25 bytes each iteration while choosing 3DES as the symmetric cipher, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext, using 6,300 queries (again for a 2048-bit modulus). This brings the overall number of queries for this variant of the attack to $900 + 16 * 4 + 6,300 = 7,264$. These parameter choices are not necessarily optimal. We give more details in Appendix 4.11.7.

4.6 Extending the attack to QUIC

DROWN can also be extended to a feasible-time man-in-the-middle attack against QUIC [79]. QUIC [37, 122] is a recent cryptographic protocol designed and implemented by Google that is intended to reduce the setup time to establish a secure connection while providing security guarantees analogous to TLS. QUIC's security relies on a static "server config" message signed by the server's public key. Jager et al. [79] observe that an attacker who can forge a signature on a malicious QUIC server config once would be able to impersonate the server indefinitely. In this section, we show an attacker with significant resources would be able to mount such an attack against a server whose RSA public keys is exposed via SSLv2.

A QUIC client receives a "server config" message, signed by the server's public key, which enumerates connection parameters: a static elliptic curve Diffie-Hellman public

Protocol	Attack type	Oracle	SSLv2 connections	Offline work	See §
TLS	Decrypt	SSLv2	$41,081$	2^{50}	4.4.3
TLS	Decrypt	Special	$7,264$	2^{51}	4.5.3
TLS	MITM	Special	$27,000$	2^{15}	4.5.2
QUIC	MITM	SSLv2	2^{25}	2^{65}	4.6.1
QUIC	MITM	Special	2^{25}	2^{25}	4.6.2
QUIC	MITM	Special	2^{17}	2^{58}	4.6.2

Table 4.3: **Summary of attacks.** "Oracle" denotes the oracle required to mount each attack, which also implies the vulnerable set of SSLv2 implementations. SSLv2 denotes any SSLv2 implementation, while "Special" denotes an OpenSSL version vulnerable to special DROWN.

value, and a validity period. In order to mount a man-in-the-middle attack against any client, the attacker wishes to generate a valid server config message containing their own Diffie-Hellman value, and an expiration date far in the future.

The attacker needs to present a forged QUIC config to the client in order to carry out a successful attack. This is straightforward, since QUIC discovery may happen over non-encrypted HTTP [69]. The server does not even need to support QUIC at all: an attacker could impersonate the attacked server over an unencrypted HTTP connection and falsely indicate that the server supports QUIC. The next time the client connects to the server, it will attempt to connect using QUIC, allowing the attacker to present the forged "server config" message and execute the attack [79].

4.6.1 QUIC signature forgery attack based on general DROWN

The attack proceeds much as in Section 4.3.2, except that some of the optimizations are no longer applicable, making the attack more expensive.

The first step is to discover a valid, PKCS conformant SSLv2 ciphertext. In the case of TLS decryption, the input ciphertext was PKCS conformant to begin with; this is not the case for the QUIC message c_0. Thus for the first phase, the attacker iterates through possible multiplier values s until they randomly encounter a valid SSLv2 message in $c_0 \cdot s^d$. For 2048-bit RSA keys, the probability of this random event is $P_{rnd} \approx 2^{-25}$; see Section 4.3.2.

		Any certificate			*Trusted certificates*		
Protocol	**Port**	**SSL/TLS**	**SSLv2 support**	**Vulnerable key**	**SSL/TLS**	**SSLv2 support**	**Vulnerable key**
SMTP	25	3,357 K	936 K (28%)	1,666 K (50%)	1,083 K	190 K (18%)	686 K (63%)
POP3	110	4,193 K	404 K (10%)	1,764 K (42%)	1,787 K	230 K (13%)	1,031 K (58%)
IMAP	143	4,202 K	473 K (11%)	1,759 K (42%)	1,781 K	223 K (13%)	1,022 K (57%)
HTTPS	443	34,727 K	5,975 K (17%)	11,444 K (33%)	17,490 K	1,749 K (10%)	3,931 K (22%)
SMTPS	465	3,596 K	291 K (8%)	1,439 K (40%)	1,641 K	40 K (2%)	949 K (58%)
SMTP	587	3,507 K	423 K (12%)	1,464 K (42%)	1,657 K	133 K (8%)	986 K (59%)
IMAPS	993	4,315 K	853 K (20%)	1,835 K (43%)	1,909 K	260 K (14%)	1,119 K (59%)
POP3S	995	4,322 K	884 K (20%)	1,919 K (44%)	1,974 K	304 K (15%)	1,191 K (60%)
(Alexa Top 1M)	443	611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

Table 4.4: **Hosts vulnerable to general DROWN** — We performed Internet-wide scans to measure the number of hosts supporting SSLv2 on several different protocols. A host is vulnerable to DROWN if its public key is exposed anywhere via SSLv2. Overall vulnerability to DROWN is much larger than support for SSLv2 due to widespread reuse of keys.

Once the first SSLv2 conformant message is found, the attacker proceeds with the signature forgery as they would in Step 2 of the TLS decryption attack. The required number of oracle queries for this step is roughly 12,468 for 2048-bit RSA keys.

Attack cost. The overall oracle query cost is dominated by the $2^{25} \approx 34$ million expected queries in the first phase, above. At a rate of 388 queries/second, the attacker would finish in one day; at a rate of 12 queries/second they would finish in one month.

For the SSLv2 export padding oracle, the offline computation to break a 40-bit symmetric key for each query requires iterating over 2^{65} keys. At our optimized GPU implementation rate of 515 million keys per second, this would require 829,142 GPU days. Our experimental GPU hardware retails for $400. An investment of $10 million to purchase 25,000 GPUs would reduce the wall clock time for the attack to 33 days.

Our implementation run on Amazon EC2 processed about 174 billion keys per `g2.2xlarge` instance-hour, so at a cost of $0.09/instance-hour the full attack would cost $9.5 million and could be parallelized to Amazon's capacity.

4.6.2 Optimized QUIC signature forgery based on special DROWN

For targeted servers that are vulnerable to special DROWN, we are unaware of a way to combine the two special DROWN oracles; the attacker would have to choose a single oracle which minimizes his subjective cost. For the Extra Clear oracle, there is only negligible computation per oracle query, so the computational cost for the first phase is 2^{25}. For the Leaky Export oracle, as explained below, the required offline work is

2^{58}, and the required number of server connections is roughly $145,573$. Both oracles appear to bring this attack well within the means of a moderately provisioned adversary.

Mounting the attack using Leaky Export. For a 2048-bit RSA modulus, the probability of a random message being conformant when querying $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is $P_{rnd} \approx (1/256)^2 * (255/256)^8 * (1-(255/256)^{246}) \approx 2^{-17}$. Therefore, to compute c^d when c is not SSLv2 conformant, the attacker randomly generates values for s and tests $c \cdot s^e$ against the Leaky Export oracle. After roughly $2^{17} \approx 131,000$ queries, they obtain a positive response, and learn that $c^d \cdot s$ starts with bytes `0x00 02`.

Naïvely, it would seem the attacker can then apply one of the techniques presented in this work, but $\mathcal{O}_{\text{SSLv2-export-leaky}}$ does not provide knowledge of any least significant plaintext bytes when the plaintext length is not at most the correct one. Instead, the attacker proceeds directly according to the algorithm presented in [23]. Referring to Table 1 in [23], $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is denoted with the term `FFT`, as it returns a positive response for a correctly padded plaintext of any length, and the median number of required queries for this oracle is 14,501. This number of queries is dominated by the 131,000 queries the attacker has already executed. As each query requires testing roughly 2^{41} keys, the required offline work is approximately 2^{58}.

Future changes to QUIC. In addition to disabling QUIC support for non-whitelisted servers, Google have informed us that they plan to change the QUIC standard, so that the "server config" message will include a client nonce to prove freshness. They also plan to limit QUIC discovery to HTTPS.

4.7 Measurements

We performed Internet-wide scans to analyze the number of systems vulnerable to DROWN. A host is directly vulnerable to general DROWN if it supports SSLv2. Similarly, a host is directly vulnerable to special DROWN if it supports SSLv2 and has the extra clear bug (which also implies the leaky export bug). These directly vulnerable hosts can be used as oracles to attack any other host with the same key. Hosts that do not support SSLv2 are still vulnerable to general or special DROWN if their RSA key pair is exposed by any general or special DROWN oracle, respectively. The oracles may be on an entirely different host or port. Additionally, any host serving a browser-trusted certificate is vulnerable to a special DROWN man-in-the-middle if any name on the certificate appears on any other certificate containing a key that is exposed by a special DROWN oracle.

We used ZMap [49] to perform full IPv4 scans on eight different ports during late

		Any certificate			*Trusted certificates*		
Protocol	**Port**	**SSL/TLS**	**Special DROWN oracles**	**Vulnerable key**	**SSL/TLS**	**Vulnerable key**	**Vulnerable name**
SMTP	25	3,357 K	855 K (25%)	896 K (27%)	1,083 K	305 K (28%)	398 K (37%)
POP3	110	4,193 K	397 K (9%)	946 K (23%)	1,787 K	485 K (27%)	674 K (38%)
IMAP	143	4,202 K	457 K (11%)	969 K (23%)	1,781 K	498 K (30%)	690 K (39%)
HTTPS	443	34,727 K	4,029 K (12%)	9,089 K (26%)	17,490 K	2,523 K (14%)	3,793 K (22%)
SMTPS	465	3,596 K	334 K (9%)	765 K (21%)	1,641 K	430 K (26%)	630 K (38%)
SMTP	587	3,507 K	345 K (10%)	792 K (23%)	1,657 K	482 K (29%)	667 K (40%)
IMAPS	993	4,315 K	892 K (21%)	1,073 K (25%)	1,909 K	602 K (32%)	792 K (42%)
POP3S	995	4,322 K	897 K (21%)	1,108 K (26%)	1,974 K	641 K (32%)	835 K (42%)
(Alexa Top 1M)	443	611 K	22 K (4%)	52 K (9%)	456 K	33 K (7%)	85 K (19%)

Table 4.5: **Hosts vulnerable to special DROWN** — A server is vulnerable to special DROWN if its key is exposed by a host with the CVE-2016-0703 bug. Since the attack is fast enough to enable man-in-the-middle attacks, a server is also vulnerable (to impersonation) if any name in its certificate is found in any trusted certificate with an exposed key.

January and February 2016. We examined port 443 (HTTPS), and common email ports 25 (SMTP with STARTTLS), 110 (POP3 with STARTTLS), 143 (IMAP with STARTTLS), 465 (SMTPS), 587 (SMTP with STARTTLS), 993 (IMAPS), and 995 (POP3S). For each open port, we attempted three complete handshakes: one normal handshake with the highest available SSL/TLS version; one SSLv2 handshake requesting an export RC2 cipher suite; and one SSLv2 handshake with a non-export cipher and sixteen bytes of plaintext key material sent during key exchange, which we used to detect if a host has the extra clear bug.

We summarize our general DROWN results in Table 4.4. The fraction of SSL/TLS hosts that directly supported SSLv2 varied substantially across ports. 28% of SMTP servers on port 25 supported SSLv2, likely due to the opportunistic encryption model for email transit. Since SMTP fails-open to plaintext, many servers are configured with support for the largest possible set of protocol versions and cipher suites, under the assumption that even bad or obsolete encryption is better than plaintext [35]. The other email ports ranged from 8% for SMTPS to 20% for POP3S and IMAPS. We found 17% of all HTTPS servers, and 10% of those with a browser-trusted certificate, are directly vulnerable to general DROWN.

OpenSSL SSLv2 cipher suite selection bug.

We discovered that OpenSSL servers do not respect the cipher suites advertised in the SSLv2 `ServerHello` message. That is, a malicious client can select an *arbitrary* cipher suite in the `ClientMasterKey` message, regardless of the contents of the `ServerHello`, and force the use of export cipher suites even if they are explicitly disabled in the server configuration. To fully detect SSLv2 oracles, we configured our scanner to ignore the `ServerHello` cipher list. The cipher selection bug helps explain the wide support for SSLv2—the protocol appeared disabled, but non-standard clients could still complete handshakes.

Widespread public key reuse. Reuse of RSA key material across hosts and certificates is widespread [72, 75]. Often this is benign: organizations may issue multiple TLS certificates for distinct domains with the same public key in order to simplify use of TLS acceleration hardware and load balancing. However, there is also evidence that system administrators may not entirely understand the role of the public key in certificates. For example, in the wake of the Heartbleed vulnerability, a substantial fraction of compromised certificates were reissued with the same public key [48].

There are many reasons why the same public key or certificate would be reused across different ports and services within an organization. For example a mail server that serves SMTP, POP3, and IMAP from the same daemon would likely share the same TLS configuration. Additionally, an organization might choose to purchase a single wildcard TLS certificate, and use it on both web servers and mail servers. Public keys have also been observed to be widely shared across independent organizations due to default certificates and public keys that are shipped with networked devices and software, improperly configured virtual machine images, and random number generation flaws.

The number of hosts vulnerable to DROWN rises significantly when we take RSA key reuse into account. For HTTPS, 17% of hosts are vulnerable to general DROWN because they support both TLS and SSLv2 on the HTTPS port, but 33% are vulnerable when considering RSA keys used by another service.

Special DROWN. As shown in Table 4.5, 9.1 M HTTPS servers (26%) are vulnerable to special DROWN, as are 2.5 M HTTPS servers with browser-trusted certificates (14%). 66% as many HTTPS hosts are vulnerable to special DROWN as to general DROWN (70% for browser-trusted servers). While 2.7 M public keys are vulnerable to general DROWN, only 1.1 M are vulnerable to special DROWN (41% as many). Vulnerability among Alexa Top Million domains is also lower, with only 9% of domains vulnerable (7% for browser-trusted domains).

Since special DROWN enables active man-in-the-middle attacks, any host serving a browser-trusted certificate with at least one name that appears on any certificate with an RSA key exposed by a special DROWN oracle is vulnerable to an impersonation attack. Extending our search to account for certificates with shared names, we find that 3.8 M (22%) hosts with browser-trusted certificates are vulnerable to man-in-the-middle attacks, as well as 19% of the browser-trusted domains in the Alexa Top Million.

4.8 Related work

TLS has had a long history of implementation flaws and protocol attacks [20, 21, 31, 44, 48, 107, 120]. We discuss relevant Bleichenbacher and cross-protocol attacks below.

Bleichenbacher's attack. Bleichenbacher's adaptive chosen ciphertext attack against SSL was first published in 1998 [32]. Several works have adapted his attack to different scenarios [23, 78, 92]. The TLS standard explicitly introduces countermeasures against the attack [42], but several modern implementations have been discovered to be vulnerable to timing-attack variants in recent years [105, 136]. These side-channel attacks are implementation failures and only apply when the attacker is co-located with the victim.

Cross-protocol attacks. Jager et al. [79] showed that a cross-protocol Bleichenbacher RSA padding oracle attack is possible against the proposed TLS 1.3 standard, in spite of the fact that TLS 1.3 does not include RSA key exchange, if server implementations use the same certificate for previous versions of TLS and TLS 1.3. Wagner and Schneier [133] developed a cross-cipher suite attack for SSLv3, in which an attacker could reuse a signed server key exchange message in a later exchange with a different cipher suite. Mavrogiannopoulos et al. [101] developed a cross-cipher suite attack allowing an attacker to use elliptic curve Diffie-Hellman as prime field Diffie-Hellman.

Attacks on export-grade cryptography. Recently, the FREAK [28] and Logjam [17] attacks allowed an active attacker to downgrade a connection to export-grade RSA and Diffie-Hellman, respectively. DROWN exploits export-grade symmetric ciphers, completing the export-grade cryptography attack trifecta.

4.9 Discussion

4.9.1 Implications for modern protocols

Although the protocol flaws in SSLv2 enabling DROWN are not present in recent TLS versions, many modern protocols meet a subset of the requirements to be vulnerable to a DROWN-style attack. For example:

1. RSA key exchange. TLS 1.2 [42] allows this.

2. Reuse of server-side nonce by the client. QUIC [37] allows this.

3. Server sends a message encrypted with the derived key before the client. QUIC, TLS 1.3 [118], and TLS False Start [93] do this.

4. Deterministic cipher parameters are generated from the premaster secret and nonces. This is the case for all TLS stream ciphers and TLS 1.0 block ciphers.

DROWN has a natural adaptation when all three properties are present. The attacker exposes a Bleichenbacher oracle by connecting to the server twice with the identical RSA ciphertexts and server-side nonces. If the RSA ciphertext is PKCS conformant, the server will respond with identical messages across both connections; otherwise they will differ.

4.9.2 Lessons for key reuse

DROWN illustrates the cryptographic principle that keys should be single use. Often, this principle is primarily applied to keys that are used to both sign and decrypt, but DROWN illustrates that using keys *for different protocol versions* can also be a serious security risk. Unfortunately, there is no widely supported way to pin X.509 certificates to specific protocols. While using per-protocol certificates may help defend against passive attacks, an active attacker could still leverage any certificate with a matching name.

4.9.3 Harms from obsolete cryptography

Recent years have seen a significant number of serious attacks exploiting outdated and obsolete cryptography. Many protocols and cryptographic primitives that were demonstrated to be weak decades ago are surprisingly common in real-world systems.

DROWN exploits a modification of an 18-year-old attack against a combination of protocols and ciphers that have long been superseded by better options: the SSLv2 protocol, export cipher suites, and PKCS #1 v1.5 RSA padding. In fact, support for RSA as a key exchange method, including the use of PKCS #1 v1.5, is mandatory even for TLS 1.2. The attack is made more severe by implementation flaws in rarely used code.

Our work serves as yet another reminder of the importance of removing deprecated technologies before they become exploitable vulnerabilities. In response to many of the vulnerabilities listed above, browser vendors have been aggressively warning end users when TLS connections are negotiated with unsafe cryptographic parameters, including SHA-1 certificates, small RSA and Diffie-Hellman parameters, and SSLv3 connections. This process is currently happening in a piecemeal fashion, primitive by primitive. Vendors and developers rightly prioritize usability and backward compatibility in standards, and are willing to sacrifice these only for practical attacks. This approach works less well for cryptographic vulnerabilities, where the first sign of a weakness, while far from being practically exploitable, can signal trouble in the future. Communication issues between academic researchers and vendors and developers have been voiced by many in the community, including Green [66] and Jager et al. [77].

The long-term solution is to proactively remove these obsolete technologies. There is movement towards this already: TLS 1.3 has entirely removed RSA key exchange and has restricted Diffie-Hellman key exchange to a few groups large enough to withstand cryptanalytic attacks long in the future. The CA/Browser forum will remove support for SHA-1 certificates this year. Resources such as the SSL Labs SSL Reports have gathered information about best practices and vulnerabilities in one place, in order to encourage administrators to make the best choices.

4.9.4 Harms from weakening cryptography

Export-grade cipher suites for TLS deliberately weakened three primitives to the point that they are now broken even to enthusiastic amateurs: 512-bit RSA key exchange, 512-bit Diffie-Hellman key exchange, and 40-bit symmetric encryption. All three deliberately weakened primitives have been cornerstones of high-profile attacks: FREAK exploits export RSA, Logjam exploits export Diffie-Hellman, and now DROWN exploits export symmetric encryption.

Like FREAK and Logjam, our results illustrate the continued harm that a legacy of deliberately weakened export-grade cryptography inflicts on the security of modern systems, even decades after the regulations influencing the original design were lifted. The attacks described in this paper are fully feasible against export cipher suites today. The technical debt induced by cryptographic “front doors” has left implementations vulnerable for decades. With the slow rate at which obsolete protocols and primitives fade away, we can expect some fraction of hosts to remain vulnerable for years to come.

4.10 Public key reuse

Reuse of RSA keys among different services was identified as a huge amplification to the number of services vulnerable to DROWN. Table 4.6 describes the number of reused RSA keys among different protocols. The two clusters 110-143 and 993-995 stick out as they share the majority of public keys. This is expected, as most of these ports are served by the same IMAP/POP3 daemon. The rest of the ports also share a substantial fraction of public keys, usually between 21% and 87%. The numbers for HTTPS (port 443) differ as there are four times as many public keys in HTTPS as in the second largest protocol.

Port	25 (SMTP)	110 (POP3)	143 (IMAP)	443 (HTTPS)	465 (SMTPS)	587 (SMTP)	993 (IMAPS)
25	1,115 (100%)	331 (32%)	318 (32%)	196 (4%)	403 (47%)	307 (48%)	369 (33%)
110	331 (30%)	1,044 (100%)	795 (79%)	152 (3%)	337 (39%)	222 (35%)	819 (72%)
143	318 (29%)	795 (76%)	1,003 (100%)	149 (3%)	321 (38%)	220 (35%)	878 (78%)
443	196 (18%)	152 (15%)	149 (15%)	4,579 (100%)	129 (15%)	94 (15%)	175 (16%)
465	403 (36%)	337 (32%)	321 (32%)	129 (3%)	857 (100%)	463 (73%)	396 (35%)
587	307 (28%)	222 (21%)	220 (22%)	94 (2%)	463 (54%)	637 (100%)	259 (23%)
993	369 (33%)	819 (78%)	878 (88%)	175 (4%)	396 (46%)	259 (41%)	1,131 (100%)
995	321 (29%)	877 (84%)	755 (75%)	151 (3%)	364 (42%)	229 (36%)	859 (76%)

Table 4.6: **Impact of key reuse across ports.** Number of shared public keys among two ports, in thousands. Each column states what number and percentage of keys from the port in the header row are used on other ports. For example, 18% of keys used on port 25 are also used on port 443, but only 4% of keys used on port 443 are also used on port 25.

4.11 Adaptations to Bleichenbacher's attack

4.11.1 Success probability of fractions

For a given fraction u/t, the success probability with a randomly chosen TLS conformant ciphertext can be computed as follows. Let m_0 be a random TLS conformant message, $m_1 = m_0 \cdot u/t$, and let ℓ_k be the expected length of the unpadded message. For $s = u/t \bmod N$ where u and t are coprime, m_1 will be SSLv2 conformant if the following conditions all hold:

1. m_0 is divisible by t. For a randomly generated m_0, this condition holds with probability $1/t$.

2. $m_1[1] = 0$ and $m_1[2] = 2$, or the integer $m \cdot u/t \in [2B, 3B)$. For a randomly generated m_0 divisible by t, this condition holds with probability

$$P = \begin{cases} 3 - 2 \cdot t/u & \text{for } 2/3 < u/t < 1 \\ 3 \cdot t/u - 2 & \text{for } 1 < u/t < 3/2 \\ 0 & \text{otherwise} \end{cases}$$

3. $\forall i \in [3, \ell_m - (\ell_k + 1)], m_1[i] \neq 0$, or all bytes between the first two bytes and the $(k+1)$ least significant bytes are non-zero. This condition holds with probability $(1 - 1/256)^{\ell_m - (\ell_k + 3)}$.

4. $m_1[\ell_m - \ell_k] = 0$: the $(\ell_k + 1)$st least significant byte is 0. This condition holds with probability $1/256$.

Using the above formulas for $u/t = 7/8$, the overall probability of success is $P = 1/8 \cdot 0.71 \cdot 0.37 \cdot 1/256 = 1/7,774$; thus the attacker expects to find an SSLv2 conformant ciphertext after testing 7,774 randomly chosen TLS conformant ciphertexts. The attacker can decrease the number of TLS conformant ciphertexts needed by multiplying each candidate ciphertext by several fractions.

Note that testing random s values until $c_1 = c_0 \cdot s^e \bmod N$ is SSLv2 conformant yields a success probability of $P_{rnd} \approx (1/256)^3 * (255/256)^{249} \approx 2^{-25}$.

4.11.2 Optimizing the chosen set of fractions

In order to deduce the validity of a single ciphertext, the attacker would have to perform a non-trivial brute-force search over all 5 byte `master_key` values. This translates into 2^{40} encryption operations.

The search space can be reduced by an additional optimization, relying on the fractional multipliers used in the first step. If the attacker uses $u/t = 8/7$ to compute a new SSLv2 conformant candidate, and m_0 is indeed divisible by $t = 7$, then the new candidate message $m_1 = m_0/t \cdot u$ is divisible by $u = 8$, and the last three bits of m_1 (and thus mk_{secret}) are zero. This allows reducing the searched `master_key` space by selecting specific fractions.

More generally, for an integer u, the largest power of 2 by which u is divisible is denoted by $v_2(u)$, and multiplying by a fraction u/t reduces the search space by a factor of $v_2(u)$. With this observation, the trade-off between the 3 metrics: the required number of intercepted ciphertexts, the required number of queries, and the required number of encryption attempts, becomes non-trivial to analyze.

Therefore, we have resorted to using simulations when evaluating the performance metrics for sets of fractions. The probability that multiplying a ciphertext by any fraction out of a given set of fractions results in an SSLv2 conformant message is difficult to compute, since the events are in fact inter-dependent: If $m \cdot 16/15$ is conforming, then m is divisible by 5, greatly increasing the probability that $m \cdot 4/5$ is also conforming. However, it is easy to perform a Monte Carlo simulation, where we randomly generate ciphertexts, and measure the probability that any fraction out of a given set produces a conforming message. The expected required number of intercepted ciphertexts is the inverse of that probability.

Formally, if we denote the set of fractions as F, and the event that a message m is conforming as $C(m)$, we perform a Monte Carlo estimation of the probability $P_F = P(\exists f \in F : C(m \cdot f))$, and the expected number of required intercepted ciphertexts equals $1/P_F$. The required number of oracle queries is simply $1/P_F \cdot |F|$. Accordingly, the required number of server connections is $2 \cdot 1/P_F \cdot |F|$, since each oracle query requires two server connections. And as for the required number of encryption attempts, if we denote this number when querying with a given fraction $f = u/t$ as E_f, then $E_f = E_{u/t} = 2^{40-v_2(u)}$. We further define the required encryption attempts when testing a ciphertext with a given set of fraction F as $E_F = \sum_{f \in F} E_f$. Then the required number of encryption attempts in Phase 1 for a given set of fractions is $(1/P_F) \cdot E_F$.

We can now give precise figures for the expected number of required intercepted ciphertexts, connections to the targeted server, and encryption attempts. The results presented in Table 4.1 were obtained using the above approach with one billion random ciphertexts per fraction set F.

4.11.3 Rotation and multiplier speedups

For a randomly chosen s, the probability that the two most significant bytes are `0x00 02` is 2^{-16}; for a 2028-bit modulus N the probability that the next $\ell_m - \ell_k - 3$ bytes of m_2 are all nonzero is about 0.37 as in the previous section, and the probability that the $\ell_k + 1$ least significant delimiter byte is `0x00` is 1/256. Thus a randomly chosen s will work with probability $2^{-25.4}$ and the attacker expects to try $2^{25.4}$ values for s before succeeding.

However, since the attacker has already learned $\ell_k + 3$ most significant bytes of $m_1 \cdot R^{-1} \bmod N$, for $\ell_k \geq 4$ and $s < 2^{30}$ they do not need to query the oracle to learn if the two most significant bytes are SSLv2 conformant; they can compute this themselves from their knowledge of $\tilde{m}_1 \cdot R^{-1}$. They iterate through values of s, test that the top two bytes of $\tilde{m}_1 \cdot R^{-1} \bmod N$ are `0x00 02`, and only query the oracle for s values that satisfy this test. Therefore, for a 2048-bit modulus they expect to test 2^{16} values offline per oracle query. The probability that a query is conformant is then $P = (1/256) * (255/256)^{249} \approx 1/678$, so they expect to perform 678 oracle queries before finding a fully SSLv2 conformant ciphertext $c_2 = (s \cdot R^{-1})^e c_1 \bmod N$.

We can speed up the brute force testing of 2^{16} values of s using algebraic lattices. We are searching for values of s satisfying $\tilde{m}_1 R^{-1} s < 3B \bmod N$, or given an offset s_0 we would like to find solutions x and z to the equation $\tilde{m}_1 R^{-1}(s_0 + x) = 2B + z \bmod N$ where $|x| < 2^{16}$ and $|z| < B$. Let $X = 2^{15}$. We can construct the lattice basis

$$L = \begin{bmatrix} -B & X\tilde{m}_1 R^{-1} & \tilde{m}_1 R^{-1} s_0 + B \\ 0 & XN & 0 \\ 0 & 0 & N \end{bmatrix}$$

We then run the LLL algorithm [95] on L to obtain a reduced lattice basis V containing vectors v_1, v_2, v_3. We then construct the linear equations $f_1(x,z) = v_{1,1}/B \cdot z + v_{1,2}/X \cdot x + v_{1,3} = 0$ and $f_2(x,z) = v_{2,1}/B \cdot z + v_{2,2}/X \cdot x + v_{2,3} = 0$ and solve the system of equations to find a candidate integer solution $x = \tilde{s}$. We then test $s = \tilde{s} + s_0$ as our candidate solution in this range.

$\det L = XZN^2$ and $\dim L = 3$, thus we expect the vectors v_i in V to have length approximately $|v_i| \approx (XZN^2)^{1/3}$. We will succeed if $|v_i| < N$, or in other words $XZ < N$. $N \approx 2^{8\ell_m}$, so we expect to find short enough vectors. This approach works well in practice and is significantly faster than iterating through 2^{16} possible values of $\tilde{s}$ for each query.

In summary, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ and we know several most significant bytes of m_2, using only a few hundred oracle queries in expectation. We can iterate this process as many times as we like to continue generating SSLv2 conformant ciphertexts c_i for which we know increasing numbers of most significant bytes, and which have a known multiplicative relationship to our original message c_0.

4.11.4 Rotations in the general DROWN attack

After the first phase, we have learned an SSLv2 conformant ciphertext c_1, and we wish to shift known plaintext bytes from least to most significant bits. Since we learn the least significant 6 bytes of plaintext of m_1 from a successful oracle $\mathcal{O}_{\textsf{SSLv2-export}}$ query, we could use a shift of 2^{-48} to transfer 48 bits of known plaintext to the most significant bits of a new ciphertext. However, we perform a slight optimization here, to reduce the number of encryption attempts. We instead use a shift of 2^{-40}, so that the least significant byte of $m_1 \cdot 2^{-40}$ and $\tilde{m_1} \cdot 2^{-40}$ will be known. This means that we can compute the least significant byte of $m_1 \cdot 2^{-40} \cdot s \bmod N$, so oracle queries now only require 2^{32} encryption attempts each. This brings the total expected number of encryption attempts for each shift to $2^{32} * 678 \approx 2^{41}$.

We perform two such plaintext shifts in order to obtain an SSLv2 conformant message, m_3 that resides in a narrow interval of length at most $2^{8\ell - 66}$. We can then obtain a multiplier s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant. Since m_3 lies in an interval of length at most $2^{8\ell - 66}$, with high probability for any $s_3 < 2^{30}$, $m_3 \cdot s_3$ lies in an interval of length at most $2^{8\ell_m - 36} < B$, so we know the two most significant bytes of $m_3 \cdot s_3$. Furthermore, we know the value of the 6 least significant bytes after multiplication. We therefore test possible values of s_3, and for values such that $m_3 \cdot s_3 \in [2B, 3B)$, and $(m_3 \cdot s_3)[\ell_m - 5] = 0$, we query the oracle with $c_3 \cdot s_3^e \bmod N$. The only condition for PKCS conformance which we haven't verified before querying the oracle is the requirement of non-zero padding, which holds with probability 0.37.

In summary, after roughly $1/0.37 = 2.72$ queries we expect a positive response from the oracle. Since we know the value of the 6 least significant bytes after multiplication, this phase does not require performing an exhaustive search. If the message is SSLv2 conformant after multiplication, we know the symmetric key, and can test whether it correctly decrypts the `ServerVerify` message.

4.11.5 Adapted Bleichenbacher iteration

After we have bootstrapped the attack using rotations, the original algorithm proposed by Bleichenbacher can be applied with minimal modifications.

The original step obtains a message that starts with the required `0x00 02` bytes once in roughly every two queries on average, and requires the number of queries to be roughly $16\ell_m$. Since we know the value of the 6 least significant bytes after multiplying by any integer, we can only query the oracle for multipliers that result in a zero 6th least significant byte, and again an exhaustive search over keys is not required. However, we cannot ensure that the padding is non-zero when querying, which again holds with probability 0.37. Therefore, for a 2048-bit modulus, the overall expected number of queries for this phase is roughly $2048 * 2/0.37 = 11,070$.

4.11.6 Special DROWN MITM performance

For the first step, the probability that the three padding bytes are correct remains unchanged. The probability that all the intermediate padding bytes are non-zero is now slightly higher, $P_1 = (1-1/256)^{229} = 0.41$, yielding an overall maximal success probability $P = 0.1 \cdot 0.41 \cdot \frac{1}{256} = 1/6,244$ per oracle query. Since the attacker now only needs to connect to the server once per oracle query, the expected number of connections in this step is the same, $6,243$. Phase 1 now yields a message with 3 known padding bytes and 24 known plaintext bytes.

For the remaining rotation steps, each rotation requires an expected 630 oracle queries. The attacker could now complete the original Bleichenbacher attack by performing 11,000 sequential queries in the final phase. However, with this more powerful oracle it is more efficient to apply a rotation 10 more times to recover the remaining plaintext bits. The number of queries required in this phase is now $10 \cdot 256/0.41 \approx 6,300$, and the queries for each of the 10 steps can be executed in parallel.

Using multiple queries per fraction. For the $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}$ oracle, the attacker can increase their chances of success by querying the server multiple times per ciphertext and fraction, using different cipher suites with different key lengths. They can negotiate DES and hope the 9th least significant byte is zero, then negotiate 128-bit RC4 and hope the 17th least significant byte is zero, then negotiate 3DES and hope the 25th least significant is zero. All three queries also require the intermediate padding bytes to be non-zero. This technique triples the success probability for a given pair of (ciphertext, fraction), at a cost of triple the queries. Its primary benefit is that fractions with smaller denominators (and thus higher probabilities of success) are now even more likely to succeed.

For a random ciphertext, when choosing 70 fractions, the probability of the first zero delimiter byte being in one of these three positions is 0.01. Hence, the attacker can use only 100 recorded ciphertexts, and expect to use $100*70*3 = 21,000$ oracle queries. For the Extra Clear oracle, each query requires one SSLv2 connection to the server. After obtaining the first positive response from the oracle, the attacker proceeds to phase 2 using 3DES.

4.11.7 Special DROWN with combined oracles

Using the Leaky Export oracle, the probability that a fraction u/t will result in a positive response is $P = P_0 * P_3$, where the formula for computing $P_0 = P((m \cdot u/t)[1,2] = 00||02)$ is provided in Appendix 4.11.1, and P_3 is, for a 2048-bit modulus:

$$\begin{aligned} P_3 = P(\texttt{0x00} \, &\notin \, \{m_3, \ldots, m_{10}\} \wedge \\ \texttt{0x00} \, &\in \, \{m_{11}, \ldots, m_\ell\}) \\ = (1-1/256)^8 &* (1-(1-1/256)^{246}) = 0.60 \end{aligned} \tag{4.1}$$

Phase 1. Our goal for this phase is to obtain a divisor t as large as possible, such that $t|m$. We generate a list of fractions, sorted in descending order of the probability of resulting in a positive response from $\mathcal{O}_{\textsf{SSLv2-export-leaky}}$. For a given ciphertext c, we then query with the 50 fractions in the list with the highest probability, until we obtain a first positive response for a fraction u_0/t_0. We can now deduce that $t_0|m$. We then generate a list of fractions u/t where t is a multiple of t_0, sort them again by success probability, and again query with the 50 most probable fractions, until a positive answer is obtained, or the list is exhausted. If a positive answer is obtained, we iteratively re-apply this process, until the list is exhausted, resulting in a final fraction u^*/t^*.

Phase 2. We then query with all fractions denominated by t^*, and hope the ciphertext decrypts to a plaintext of one of seven possible lengths: $\{2,3,4,5,8,16,24\}$. Assuming that this is the case, we learn at least three least significant bytes, which allows us to use the shifting technique in order to continue the attack. Detecting plaintext lengths 8, 16 and 24 can be accomplished using three Extra Clear oracle queries, employing DES, 128-bit RC4 and 3DES, respectively, as the chosen cipher suite. Detecting plaintext lengths 2, 3, 4 and 5 can be accomplishing by using a single Leaky Export oracle query, which requires at most 2^{41} offline computation. In fact, the optimization over the key search space described in Section 4.3.2.1 is applicable here and can slightly reduce the required computation. Therefore, by initiating four SSLv2 connections and performing at most 2^{41} offline work, the attacker can test for ciphertexts which decrypt to one of these seven lengths.

In practice, choosing 50 fractions per iteration as described above results in a success probability of 0.066 for a single ciphertext. Hence, the expected number of required ciphertexts is merely $1/0.066 = 15$. The expected number of fractions per ciphertext for phase 1 is 60, as in most cases phase 1 consists of just a few successful iterations. Since each fraction requires a single query to $\mathcal{O}_{\textsf{SSLv2-export-leaky}}$, the overall number of queries for this stage is $15 * 60 = 900$, and the required offline computation is at most $900 * 2^{41} \approx 2^{51}$, which is similar to general DROWN. For a 2048-bit RSA modulus, the expected number of queries for phase 2 is 16. Each query consists of three queries to $\mathcal{O}_{\textsf{SSLv2-extra-clear}}$ and one query to $\mathcal{O}_{\textsf{SSLv2-export-leaky}}$, which requires at most 2^{41} computation. Therefore in expectancy the attacker has to perform 2^{45} offline computation for phase 2.

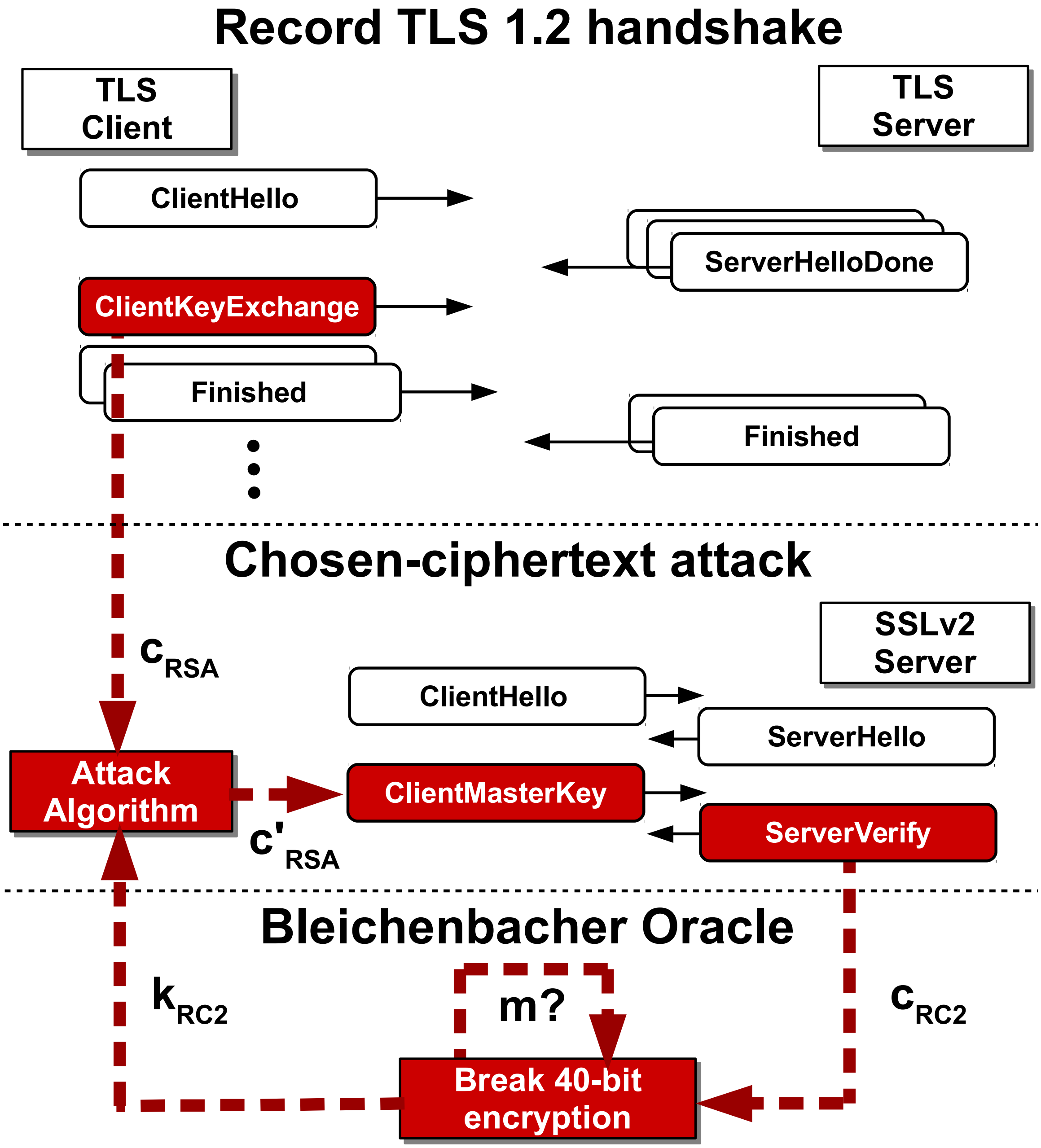

Figure 4.3: **SSLv2-based Bleichenbacher attack on TLS** — An attacker passively collects RSA ciphertexts from a TLS 1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

CHAPTER 5

Subgroup

In order for the discrete log problem $\bmod p$ to be hard, Diffie-Hellman parameters must be chosen carefully. A typical recommendation is that p should be a "safe" prime, that is, that $p = 2q + 1$ for some prime q, and that g should generate the group of order q modulo p. For p that are not safe, the group order q can be much smaller than p. For security, q must still be large enough to thwart known attacks, which for prime q run in time $O(\sqrt{q})$. A common parameter choice is to use a 160-bit q with a 1024-bit p or a 224-bit q with a 2048-bit p, to match the security level under different cryptanalytic attacks. Diffie-Hellman parameters with p and q of these sizes were suggested for use and standardized in DSA signatures [109]. For brevity, we will refer to these non-safe primes as DSA primes, and to groups using DSA primes with smaller values of q as DSA groups.

A downside of using DSA primes instead of safe primes for Diffie-Hellman is that implementations must perform additional validation checks to ensure the key exchange values they receive from the other party are contained in the correct subgroup modulo p. The validation consists of performing an extra exponentiation step. If implementations fail to validate, a 1997 attack of Lim and Lee [97] can allow an attacker to recover a static exponent by repeatedly sending key exchange values that are in very small subgroups. We describe several variants of small subgroup confinement attacks that allow an attacker with access to authentication secrets to mount a much more efficient man-in-the-middle attack against clients and servers that do not validate group orders. Despite the risks posed by these well-known attacks on DSA groups, NIST SP 800-56A, "Recommendations for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography" [24] specifically recommends DSA group parameters for Diffie-Hellman, rather than recommending using safe primes. RFC 5114 [96] includes several DSA groups for use in IETF standards.

We observe that few Diffie-Hellman implementations actually validate subgroup orders, in spite of the fact that small subgroup attacks and countermeasures are well-known and specified in every standard suggesting the use of DSA groups for Diffie-Hellman, and DSA groups are commonly implemented and supported in popular protocols. For some protocols, including TLS and SSH, that enable the server to unilaterally specify the group used for key exchange, this validation step is not possible for clients to perform with DSA primes—there is no way for the server to communicate to the client the intended order of the group. Many standards involving DSA groups further suggest that the order of the subgroup should be matched to the length of the private exponent. Using shorter private exponents yields faster exponentiation times, and is a commonly implemented optimization. However, these standards provide no security justification for decreasing the size of the subgroup to match the size of the exponents, rather than using as large a subgroup as possible. We discuss possible motivations for these recommendations later in the paper.

We conclude that adopting the Diffie-Hellman group recommendations from RFC 5114 and NIST SP 800-56A may create vulnerabilities for organizations using existing cryptographic implementations, as many libraries allow user-configurable groups but have unsafe default behaviors. This highlights the need to consider developer usability and implementation fragility when designing or updating cryptographic standards.

Our Contributions We study the implementation landscape of Diffie-Hellman from several perspectives and measure the security impact of the widespread failure of implementations to follow best security practices:

- We summarize the concrete impact of small-subgroup confinement attacks and small subgroup key recovery attacks on TLS, IKE, and SSH handshakes.
- We examined the code of a wide variety of cryptographic libraries to understand their implementation choices. We find feasible full private exponent recovery vulnerabilities in OpenSSL and the Unbound DNS resolver, and a partial private exponent recovery vulnerability for the parameters used by the Amazon Elastic Load Balancer. We observe that *no* implementation that we examined validated group order for subgroups of order larger than two by default prior to January 2016, leaving users potentially vulnerable to small subgroup confinement attacks.
- We performed Internet-wide scans of HTTPS, POP3S, SMTP with STARTTLS, SSH, IKEv1, and IKEv2, to provide a snapshot of the deployment of DSA groups and other non-“safe” primes for Diffie-Hellman, quantify the incidence of repeated public exponents in the wild, and quantify the lack of validation checks even for safe primes.
- We performed a best-effort attempt to factor $p-1$ for all non-safe primes that we found in the wild, using ˜100,000 core-hours of computation. Group 23 from RFC 5114, a 2048-bit prime, is particularly vulnerable to small subgroup key recovery

attacks; for TLS a full key recovery requires 2^{33} online work and 2^{47} offline work to recover a 224-bit exponent.

Disclosure and Mitigations We reported the small subgroup key recovery vulnerability to OpenSSL in January 2016 [123]. OpenSSL issued a patch to add additional validation checks and generate single-use private exponents by default [11]. We reported the Amazon load balancer vulnerability in November 2015. Amazon responded to our report informing us that they have removed Diffie-Hellman from their recommmended ELB security policy, and have reached out to their customers to recommend that they use these latest policies. Based on scans performed in February and May 2016, 88% of the affected hosts appear to have corrected their exponent generation behavior. We found several libraries that had vulnerable combinations of behaviours, including Unbound DNS, GnuTLS, LibTomCrypt, and Exim. We disclosed to the developers of these libraries. Unbound issued a patch, GnuTLS acknowledged the report but did not patch, and LibTomCrypt did not respond. Exim responded to our bug report stating that they would use their own generated Diffie-Hellman groups by default, without specifying subgroup order for validation [138, 140]. We found products from Cisco, Microsoft, and VMWare lacking validation that key exchange values were in the range $(1, p-1)$. We informed these companies, and discuss their responses in Section 5.2.4.

5.1 Background

5.1.1 Groups, orders, and generators

The two types of groups used for Diffie-Hellman key exchange in practice are multiplicative groups over finite fields ("mod p") and elliptic curve groups. We focus on the "mod p" case, so a group is typically specified by a prime p and a generator g, which generates a multiplicative subgroup modulo p. Optionally, the group order q can be specified; this is the smallest positive integer q satisfying $g^q \equiv 1 \bmod p$. Equivalently, it is the number of distinct elements of the subgroup $\{g, g^2, g^3, \dots \bmod p\}$.

By Lagrange's theorem, the order q of the subgroup generated by g modulo p must be a divisor of $p-1$. Since p is prime, $p-1$ will be even, and there will always be a subgroup of order 2 generated by the element -1. For the other factors q_i of $p-1$, there are subgroups of order q_i mod p. One can find a generator g_i of a subgroup of order q_i using a randomized algorithm: try random integers h until $h^{(p-1)/q_i} \neq 1 \bmod p$; $g_i = h^{(p-1)/q_i} \bmod p$ is a generator of the subgroup. A random h will satisfy this property with probability $1 - 1/q_i$.

In theory, neither p nor q is required to be prime. Diffie-Hellman key exchange is possible with a composite modulus and with a composite group order. In such cases, the order of the full multiplicative group modulo p is $\phi(p)$ where ϕ is Euler's totient function, and the order of the subgroup generated by g must divide $\phi(p)$. Outside of implementation mistakes, Diffie-Hellman in practice is done modulo prime p.

5.1.2 Diffie-Hellman Key Exchange

Diffie-Hellman key exchange allows two parties to agree on a shared secret in the presence of an eavesdropper [43]. Alice and Bob begin by agreeing on shared parameters (prime p, generator g, and optionally group order q) for an algebraic group. Depending on the protocol, the group may be requested by the initiator (as in IKE), unilaterally chosen by the responder (as in TLS), or fixed by the protocol itself (SSH originally built in support for a single group).

Having agreed on a group, Alice chooses a secret $x_a < q$ and sends Bob $y_a = g^{x_a} \bmod p$. Likewise, Bob chooses a secret $x_b < q$ and sends Alice $y_b = g^{x_b} \bmod p$. Each participant then computes the shared secret key $g^{x_a x_b} \bmod p$.

Depending on the implementation, the public values y_a and y_b might be *ephemeral*—freshly generated for each connection—or *static* and reused for many connections.

5.1.3 Discrete log algorithms

The best known attack against Diffie-Hellman is for the eavesdropper to compute the the private exponent x by calculating the discrete log of one of Alice or Bob's public value y. With knowledge of the exponent, the attacker can trivially compute the shared secret. It is not known in general whether the hardness of computing the shared secret from the public values is equivalent to the hardness of discrete log.

The *computational Diffie-Hellman assumption* states that computing the shared secret $g^{x_a x_b}$ from g^{x_a} and g^{x_b} is hard for some choice of groups. A stronger assumption, the *decisional Diffie-Hellman problem*, states that given g^{x_a} and g^{x_b}, the shared secret $g^{x_a x_b}$ is computationally indistinguishable from random for some groups. This assumption is often not true for groups used in practice; even with safe primes as defined below, many implementations use a generator that generates the full group of order $p-1$, rather than the subgroup of order $(p-1)/2$. This means that a passive attacker can always learn the value of the secret exponent modulo 2. To avoid leaking this bit of information about the exponent, both sides could agree to compute the shared secret as $y^{2x} \bmod p$. We have not seen implementations with this behavior.

There are several families of discrete log algorithms, each of which apply to special types of groups and parameter choices. Implementations must take care to avoid choices vulnerable to any particular algorithm. These include:

Small-order groups The Pollard rho [116] and Shanks' baby step-giant step algorithms [125] each can be used to compute discrete logs in groups of order q in time $O(\sqrt{q})$. To avoid being vulnerable, implementations must choose a group order with bit length at least twice the desired bit security of the key exchange. In practice, this means that group orders q should be at least 160 bits for an 80-bit security level.

Composite-order groups If the group order q is a composite with prime factorization $q = \prod_i q_i^{e_i}$, then the attacker can use the Pohlig-Hellman algorithm [114] to compute a discrete log in time $O(\sum_i e_i \sqrt{q_i})$. The Pohlig-Hellman algorithm computes the discrete log in each subgroup of order $q_i^{e_i}$ and then uses the Chinese remainder theorem to reconstruct the log modulo q. Adrian et al. [17] found several thousand TLS hosts using primes with composite-order groups, and were able to compute discrete logs for several hundred Diffie-Hellman key exchanges using this algorithm. To avoid being vulnerable, implementations should choose g so that it generates a subgroup of large prime order modulo p.

Short exponents If the secret exponent x_a is relatively small or lies within a known range of values of a relatively small size, m, then the Pollard lambda "kangaroo" algorithm [117] can be used to find x_a in time $O(\sqrt{m})$. To avoid this attack, implementations should choose secret exponents to have bit length at least twice the desired security level. For example, using a 256-bit exponent for for a 128-bit security level.

Small prime moduli When the subgroup order is not small or composite, and the prime modulus p is relatively large, the fastest known algorithm is the number field sieve [63], which runs in subexponential time in the bit length of p, $\exp\left((1.923 + o(1))(\log p)^{1/3}(\log \log p)^{2/3}\right)$. Adrian et al. recently applied the number field sieve to attack 512-bit primes in about 90,000 core-hours [17], and they argue that attacking 1024-bit primes—which are widely used in practice—is within the resources of large governments. To avoid this attack, current recommendations call for p to be at least 2048 bits [24]. When selecting parameters, implementers should ensure all attacks take at least as long as the number field sieve for their parameter set.

5.1.4 Diffie-Hellman group characteristics

"Safe" primes In order to maximize the size of the subgroup used for Diffie-Hellman, one can choose a p such that $p = 2q + 1$ for some prime q. Such a p is called a "safe" prime, and q is a Sophie Germain prime. For sufficiently large safe primes, the best attack will be solving the discrete log using the number field sieve. Many standards explicitly specify the use of safe primes for Diffie-Hellman in practice. The Oakley protocol [112] specified five "well-known" groups for Diffie-Hellman in 1998. These included three safe primes of size

768, 1024, and 1536 bits, and was later expanded to include six more groups in 2003 [89]. The Oakley groups have been built into numerous other standards, including IKE [70] and SSH [135].

DSA groups The DSA signature algorithm [109] is also based on the hardness of discrete log. DSA parameters have a subgroup order q of much smaller size than p. In this case $p-1 = qr$ where q is prime and r is a large composite, and g generates a group of order q. FIPS 186-4 [109] specifies 160-bit q for 1024-bit p and 224- or 256-bit q for 2048-bit p. The small size of the subgroup allows the signature to be much shorter than the size of p.

5.1.5 DSA Group Standardization

DSA-style parameters have also been recommended for use for Diffie-Hellman key exchange. NIST Special Publication 800-56A, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography" [24], first published in 2007, specifies that finite field Diffie-Hellman should be done over a prime-order subgroup q of size 160 bits for a 1024-bit prime p, and a 224- or 256-bit subgroup for a 2048-bit prime. While the order of the multiplicative subgroups is in line with the hardness of computing discrete logs in these subgroups, no explanation is given for recommending a subgroup of precisely this size rather than setting a minimum subgroup size or using a safe prime. Using a shorter exponent will make modular exponentiation more efficient, but the order of the subgroup q does not increase efficiency—on the contrary, the additional modular exponentiation required to validate that a received key exchange message is contained in the correct subgroup will render key exchange with DSA primes less efficient than using a "safe" prime for the same exponent length. Choosing a small subgroup order is not known to have much impact on other cryptanalytic attacks, although the number field sieve is somewhat (not asymptotically) easier as the linear algebra step is performed modulo the subgroup order q. [17]

RFC 5114, "Additional Diffie-Hellman Groups for Use with IETF Standards" [96], specifies three DSA groups with the above orders "for use in IKE, TLS, SSH, etc." These groups were taken from test data published by NIST [2]. They have been widely implemented in IPsec and TLS, as we will show below. We refer to these groups as Group 22 (1024-bit group with 160-bit subgroup), Group 23 (2048-bit group with 224-bit subgroup), and Group 24 (2048-bit group with 256-bit subgroup) throughout the remainder of the paper to be consistent with the group numbers assigned for IKE.

RFC 6989, "Additional Diffie-Hellman Tests for the Internet Key Exchange Protocol Version 2 (IKEv2)" [126], notes that "mod p" groups with small subgroups can be vulnerable to small subgroup attacks, and mandates that IKE implementations should validate that the received value is in the correct subgroup or never repeat exponents.

5.1.6 Small subgroup attacks

Since the security of Diffie-Hellman relies crucially on the group parameters, implementations can be vulnerable to an attacker who provides maliciously generated parameters that change the properties of the group. With the right parameters and implementation decisions, an attaker may be able to efficiently determine the Diffie-Hellman shared secret. In some cases, a passive attacker may be able to break a transcript offline.

Small subgroup confinement attacks In a small subgroup confinement attack, an attacker (either a man-in-the-middle or a malicious client or server) provides a key-exchange value y that lies in a subgroup of small order. This forces the other party's view of the shared secret, y^x, to lie in the subgroup generated by the attacker. This type of attack was described by van Oorschot and Wiener [131] and ascribed to Vanstone and Anderson and Vaudenay [22]. Small subgroup confinement attacks are possible even when the server does not repeat exponents—the only requirement is that an implementation does not validate that received Diffie-Hellman key exchange values are in the correct subgroup.

When working mod p, there is always a subgroup of order 2, since $p-1$ is even. A malicious client Mallory could initiate a Diffie-Hellman key exchange value with Alice and send her the value $y_M = p-1 \equiv -1 \bmod p$, which is is a generator of the group of order 2 mod p. When Alice attempts to compute her view of the shared secret as $k_a = y_M^a \bmod p$, there are only two possible values, 1 and $-1 \bmod p$.

The same type of attack works if $p-1$ has other small factors q_i. Mallory can send a generator g_i of a group of order q_i as her Diffie-Hellman key exchange value. Alice's view of the shared secret will be an element of the subgroup of order q_i. Mallory then has a $1/q_i$ chance of blindly guessing Alice's shared secret in this invalid group. Given a message from Alice encrypted using Alice's view of the shared secret, Mallory can brute force Alice's shared secret in q_i guesses.

More recently, Bhargavan and Delignat-Lavaud [29] describe "key synchronization" attacks against IKEv2 where a man-in-the-middle connects to both the initiator and responder in different connections, uses a small subgroup confinement attack against both, and observes that there is a $1/q_i$ probability of the shared secrets being the same in both connections. Bhargavan and Leurent [31] describe several attacks that use subgroup confinement attacks to obtain a transcript collision and break protocol authentication.

To protect against subgroup confinement attacks, implementations should use prime-order subgroups with known subgroup order. Both parties must validate that the key exchange values they receive are in the proper subgroup. That is, for a known subgroup order q, a received Diffie-Hellman key exchange value y should satisfy $y^q \equiv 1 \bmod p$. For a safe prime, it suffices to check that y is strictly between 1 and $p-1$.

Small subgroup key recovery attacks Lim and Lee [97] discovered a further attack that arises when an implementation fails to validate subgroup order and resues a static secret exponent for multiple key exchanges. A malicious party may be able to perform multiple subgroup confinement attacks for different prime factors q_i of $p-1$ and then use the Chinese remainder theorem to reconstruct the static secret exponent.

The attack works as follows. Let $p-1$ have many small factors $p-1 = q_1 q_2 \dots q_n$. Mallory, a malicious client, uses the procedure described in Section 5.1.1 to find a generator of the subgroup g_i of order q_i mod p. Then Mallory transmits g_i as her Diffie-Hellman key exchange value, and receives a message encrypted with Alice's view of the shared secret $g_i^{x_a}$, which Mallory can brute force to learn the value of x_a mod q_i. Once Mallory has repeated this process several times, she can use the Chinese remainder theorem to reconstruct x_a mod $\prod_i q_i$. The running time of this attack is $\sum_i q_i$, assuming that Mallory performs an offline brute-force search for each subgroup.

A randomly chosen prime p is likely to have subgroups of large enough order that this attack is infeasible to carry out for all subgroups. However, if in addition Alice's secret exponent x_a is small, then Mallory only needs to carry out this attack for a subset of subgroups of orders $q_1, \dots, q_k$ satisfying $\prod_{i=0}^{k} q_i > x_a$, since the Chinese remainder theorem ensures that x_a will be uniquely defined. Mallory can also improve on the running time of the attack by taking advantage of the Pollard lambda algorithm. That is, she could use a small subgroup attack to learn the value of x_a mod $\prod_{i=1}^{k} q_i$ for a subset of subgroups $\prod_{i=1}^{k} q_i < x_a$, and then use the Pollard lambda algorithm to reconstruct the full value of a, as it has now been confined to a smaller interval.

In summary, an implementation is vulnerable to small subgroup key recovery attacks if it does not verify that received Diffie-Hellman key exchange values are in the correct subgroup; uses a prime p such that $p-1$ has small factors; and reuses Diffie-Hellman secret exponent values. The attack is made even more practical if the implementation uses small exponents.

A related attack exists for elliptic curve groups: an invalid curve attack. Similarly to the case we describe above, the attacker generates a series of elliptic curve points of small order and sends these points as key exchange messages to the victim. If the victim does not validate that the received point is on the intended curve, they return a response that reveals information about the secret key modulo different group orders. After enough queries, the attacker can learn the victim's entire secret. Jager, Schwenk, and Somorovsky [80] examined eight elliptic curve implementations and discovered two that failed to validate the received curve point. For elliptic curve groups, this attack can be much more devastating because

Application	**Crypto Library**	**Short Exponent**	**Exponent Reuse**
OpenSSH	OpenSSL	No	No
Cerberus	OpenSSL	No	Yes
GNU lsh	GnuTLS	No	No
Dropbear	LibTomCrypt	No	No
Lighttpd	OpenSSL	Yes	No
Unbound	OpenSSL	Yes	Yes
Exim	OpenSSL	Library dependent	Yes
Postfix	OpenSSL	No	No

Table 5.1: **Common application behavior** — Applications make a diverse set of decisions on how to handle Diffie-Hellman exponents, likely due to the plethora of conflicting, confusing, and incorrect recommendations available.

the attacker has much more freedom in generating different curves, and can thus find many different small prime order subgroups. For the finite field Diffie-Hellman attack, the attacker is limited only to those subgroups whose orders are factors of $p-1$.

5.2 TLS

TLS (Transport Layer Security) is a transport layer protocol designed to provide confidentiality, integrity and (most commonly) one-side authentication for application sessions. It is widely used to protect HTTP and mail protocols.

A TLS client initiates a TLS handshake with the `ClientHello` message. This message includes a list of supported cipher suites, and a client random nonce r_c. The server responds with a `ServerHello` message containing the chosen cipher suite and server random nonce r_s, and a `Certificate` message that includes the server's X.509 certificate. If the server selects a cipher suite using ephemeral Diffie-Hellman key exchange, the server additionally sends a `ServerKeyExchange` message containing the server's choice of Diffie-Hellman parameters p and g, the server's Diffie-Hellman public value $y_s = g^{x_s} \bmod p$, a signature by the server's private key over both the client and server nonces (r_c and r_s), and the server's Diffie-Hellman parameters (p, g, and y_s). The client then verifies the signature using the public key from the server's certificate, and responds with a `ClientKeyExchange` message containing the client's Diffie-Hellman public value $y_c = g^{x_c} \bmod p$. The Diffie-Hellman shared secret $Y = g^{x_s x_c} \bmod p$ is used to derive encryption and MAC keys. The client then sends `ChangeCipherSpec` and `Finished` messages. The `Finished` message contains a hash of the handshake transcript, and is encrypted and authenticated using the derived encryption and MAC keys. Upon decrypting and authenticating this message, the server verifies that the hash of the transcript matches the expected hash. Provided the hash matches, the server then sends its own `ChangeCipherSpec` and `Finished` messages, which the client then verifies. If either side fails to decrypt or authenticate the `Finished` messages, or if the transcript hashes do not match, the connection fails immediately [42].

TLS also specifies a mode of using Diffie-Hellman with fixed parameters from the server's certificate [115]. This mode is not forward secret, was never widely adopted, and has been removed from all modern browsers due to dangerous protocol flaws [74]. The only widely used form of Diffie-Hellman in TLS today is ephemeral Diffie-Hellman, described above.

5.2.1 Small Subgroup Attacks in TLS

Small subgroup confinement attacks A malicious TLS server can perform a variant of the small subgroup attack against a client by selecting group parameters g and p such that g generates an insecure group order. TLS versions prior to 1.3 give the server complete liberty to choose the group, and they do not include any method for the server to specify the desired group order q to the client. This means a client has no feasible way to validate that the group sent by the server has the desired level of security or that a server's key exchange value is in the correct group for a non-safe prime.

Similarly, a man in the middle with knowledge of the server's long-term private signing key can use a small subgroup confinement attack to more easily compromise perfect forward secrecy, without having to rewrite an entire connection. The attack is similar to the those described by Bhargavan and Delignat-Lavaud [29]. The attacker modifies the server key exchange message, leaving the prime unchanged, but substituting a generator g_i of a subgroup of small order q_i for the group generator and g_i for the server's key exchange value y_s. The attacker then forges a correct signature for the modified server key exchange message and passes it to the client. The client then responds with a client key exchange message $y_c = g_i^{x_c} \bmod p$, which the man-in-the-middle leaves unchanged. The server's view of the shared secret is then $g_i^{x_c x_s} \bmod p$, and the client's view of the shared secret is $g_i^{x_c} \bmod p$. These views are identical when $x_s \equiv 1 \bmod q_i$, so this connection will succeed with probability $1/q_i$. For small enough q_i, this enables a man in the middle to use a compromised server signing key to decrypt traffic from forward-secret ciphersuites with a reasonable probability of success, while only requiring tampering with a single handshake message, rather than having to actively rewrite the entire connection for the duration of the session.

Furthermore, if the server uses a static Diffie-Hellman key exchange value, then the attacker can perform a small subgroup key-recovery attack as the client in order to learn the server's static exponent $x_s \bmod q_i$ for the small subgroup. This enables the attacker to

Implementation	RFC 5114 Support	Allows Short Exponents	Reuses Exponents	Validates Su
Mozilla NSS	No	Yes, hardcoded	No	$g \leq 2$
OpenJDK	No	Yes, uses max of p_size / 2 and 384	No	$g \leq 2$
OpenSSL 1.0.2	Yes	Yes, if q set or if user sets a shorter length	Default until Jan '16	Yes, as of Jan
BouncyCastle	Yes	No	Application dependent	$g \leq 2$
Cryptlib	No	Yes, uses quadratic curve calculation	Application dependent	$g \leq 2$
libTomCrypt	No	Yes, hardcoded	Application dependent	No
CryptoPP	No	Yes, uses work factor calculation	Application dependent	No
Botan	Yes	Yes, uses work factor calculation	No	No
GnuTLS	Application dependent	Yes, restricts to q_size (max 256)	Application dependent	$g \leq 2$

Table 5.2: **TLS Library Behavior** — We examined popular TLS libraries to determine which weaknesses from Section 5.1.6 were present. Reuse of exponents often depends on the use of the library; the burden is on the application developer to appropriately regenerate exponents. Botan and libTomCrypt both hardcode their own custom groups, while GnuTLS allows users to specify their own parameters.

calculate a custom generator such that the client and server views of the shared secret are always identical, raising the above attack to a 100% probability of success.

Small subgroup key recovery attacks In TLS, the client must authenticate the handshake before the server, by providing a valid `Finished` message. This forces a small subgroup key recovery attack against TLS to be primarily online. To perform a Lim-Lee small subgroup key recovery attack against a server static exponent, a malicious client initiates a TLS handshake and sends a generator g_i of a small subgroup of order q_i as its client key exchange message y_c. The server will calculate $Y_s = g_i^{x_s} \bmod p$ as the shared secret. The server's view of the shared secret is confined to the subgroup of order q_i. However, since g_i and g generate separate subgroups, the server's public value $y_s = g_s^x$ gives the attacker no information about the value of the shared secret Y_s. Instead, the attacker must guess a value for $x_s \bmod q_i$, and send the corresponding client `Finished` message. If the server continues the handshake, the attacker learns that the guess is correct. Therefore, assuming the server is reusing a static value for x_s, the attacker needs to perform at most q_i queries to learn the server's secret $x_s \bmod q_i$ [97]. This attack is feasible if q_i is small enough and the server reuses Diffie-Hellman exponents for sufficiently many requests.

The attacker repeats this process for many different primes q_i, and uses the Chinese remainder theorem to combine them modulo the product of the primes q_i. The attacker can also use the Pollard lambda algorithm to reconstruct any remaining bits of the exponent [97].

We note that the TLS False Start extension allows the server to send application data before receiving the client's authentication [93]. The specification only allows this behavior for abbreviated handshakes, which do not include a full key exchange. If a full key exchange were allowed, the fact that the server authenticates first would allow a malicious client to mount a mostly offline key recovery attack.

5.2.2 OpenSSL

Prior to early 2015, OpenSSL defaulted to using static-ephemeral Diffie-Hellman values. Server applications generate a fresh Diffie-Hellman secret exponent on startup, and reuse this exponent until they are restarted. A server would be vulnerable to small subgroup attacks if it chose a DSA prime, explicitly configured the `dh->length` parameter to generate a short exponent, and failed to set `SSL_OP_SINGLE_DH_USE` to prevent repeated exponents. OpenSSL provides some test code for key generation which configures DSA group parameters, sets an exponent length to the group order, and correctly sets the `SSL_OP_SINGLE_DH_USE` to generate new exponents on every connection. We found this test code widely used across many applications. We discovered that Unbound, a DNS resolver, used the same parameters as the tests, but without setting `SSL_OP_SINGLE_DH_USE`, rendering them vulnerable to a key recovery attack. A number of other applications including Lighttpd used the same or similar code with non-safe primes, but correctly set `SSL_OP_SINGLE_DH_USE`.

In spring 2015, OpenSSL added explicit support for RFC 5114 groups [7], including the ability for servers to specify a subgroup order in a set of Diffie-Hellman group parameters. When the subgroup order is specified, the exponent length is automatically adjusted to match the subgroup size. However, the update did not contain code to validate subgroup order for key exchange values, leaving OpenSSL users vulnerable to precisely the key recovery attack outlined in Section 5.2.1.

We disclosed this vulnerability to OpenSSL in January 2016. The vulnerability was patched by including code to validate subgroup order when a subgroup was specified in a set of Diffie-Hellman parameters and setting `SSL_OP_SINGLE_DH_USE` by default [14]. Prior to this patch, any code using OpenSSL for DSA-style Diffie-Hellman parameters was vulnerable to small subgroup attacks by default.

Exim [139], a popular mail server that uses OpenSSL, provides a clear example of the fragile situation created by this update. By default, Exim uses the RFC 5114 Group 23 parameters with OpenSSL, does not set an exponent length, and does not set `SSL_OP_SINGLE_DH_USE`. In a blog post, an Exim developer explains that because of “numerous issues with automatic generation of DH parameters”, they added support for fixed groups specified in RFCs and picked Group 23 as the default [140]. Exim narrowly avoided being fully vulnerable to a key recovery attack by not including the size of the subgroup generated by q in the Diffie-Hellman parameters that it passes to OpenSSL. Had this been included,

			Number of hosts that use…			
Protocol	**Scan Date**	**Total Hosts**	**Diffie-Hellman**	**Non-Safe Primes**	**Static Exponents**	**Static Exponents and Non-Safe Primes**
HTTPS	2/2016	40,578,754	10,827,565	1,661,856	964,356	309,891
POP3S	10/2015	4,368,656	3,371,616	26,285	32,215	25
STARTTLS	10/2015	3,426,360	3,036,408	1,186,322	30,017	932
SSH	10/2015	15,226,362	10,730,527	281	1,147	0
IKEv1	2/2016	2,571,900	2,571,900	340,300	109	0
IKEv2	2/2016	1,265,800	1,265,800	177,000	52	0

Table 5.3: **IPv4 non-safe prime and static exponent usage** — Although non-safe primes see widespread use across most protocols, only a small number of hosts reuse exponents and use non-safe primes; these hosts are prime candidates for a small subgroup key recovery attack.

OpenSSL would have automatically shortened the exponent length, leaving the server fully vulnerable to a key recovery attack. For this group, an attacker can recover 130 bits of information about the secret exponent using 2^{33} online queries, but this does not allow the attacker to recover the server's 2048-bit exponent modulo the correct 224-bit group order q as the small subgroup orders q_i are all relatively prime to q.

We looked at several other applications as well, but did not find them to be vulnerable to key recovery attacks (Table 5.1).

5.2.3 Other Implementations

We examined the source code of multiple TLS implementations (Table 5.2). Prior to January 2016, no TLS implementations that we examined validated group order, even for the well-known DSA primes from RFC 5114, leaving them vulnerable to small subgroup confinement attacks.

Most of the implementations we examined attempt to match exponent length to the perceived strength of the prime. For example, Mozilla Network Security Services (NSS), the TLS library used in the Firefox browser and some versions of Chrome [8, 57], uses NIST's "comparable key strength" recommendations on key management [24] to determine secret exponent lengths from the length of the prime. [3] Thus NSS uses 160-bit exponents with a 1024-bit prime, and 224-bit exponents with a 2048-bit prime. In fall 2015, NSS added an additional check to ensure that the shared secret $g^{x_a x_b} \not\equiv 1 \bmod p$ [6].

Several implementations go to elaborate lengths to match exponent length to perceived prime strength. The Cryptlib library fits a quadratic curve to the small exponent attack cost table in the original van Oorschot paper [131] and uses the fitted curve to determine safe key lengths [67]. The Crypto++ library uses an explicit “work factor” calculation, evaluating the function $2.4n^{1/3}(\log n)^{2/3}$ [81]. Subgroup order and exponent lengths are set to twice the calculated work factor. The work factor calculation is taken from a 1995 paper by Odlyzko on integer factorization [110]. Botan, a C++ cryptography and TLS library, uses a similar work factor calculation, derived from RFC 3766 [68], which describes best practices as of 2004 for selecting public key strengths when exchanging symmetric keys. RFC 3766 uses a similar work factor algorithm to Odlyzko, intended to model the running time of the number-field sieve. Botan then doubles the length of the work factor to obtain subgroup and exponent lengths [10].

5.2.4 Measurements

We used ZMap [49] to probe the public IPv4 address space for hosts serving three TLS-based protocols: HTTPS, SMTP+STARTTLS, and POP3S. To determine which primes servers were using, we sent a `ClientHello` message containing only ephemeral Diffie-Hellman cipher suites. We combined this data with scans from Censys [45] to determine the overall population. The results are summarized in Table 5.3.

In August 2016, we conducted additional scans of a random 1% sample of HTTPS hosts on the Internet. First, we checked for nontrivial small subgroup attack vulnerability. For servers that sent us a prime p such that $p-1$ was divisible by 7, we attempted a handshake using a client key exchange value of $g_7 \bmod p$, where g_7 is a generator of a subgroup of order 7. (7 is the smallest prime factor of $p-1$ for Group 22.) When we send g_7, we expect to correctly guess the `PreMasterSecret` and complete the handshake with one seventh of hosts that do not validate subgroup order. In our scan, we were able to successfully complete a handshake with 1477 of 10714 hosts that offered a prime such that $p-1$ was

Key Exchange Value	**Support DHE**	**Accepted**
$0 \bmod p$	143.5 K	87
$1 \bmod p$	142.2 K	4.9 K
$-1 \bmod p$	143.5 K	7.6 K
$g_7 \bmod p$	10.7 K	1.5 K

Table 5.4: **TLS key exchange validation** — We performed a 1% HTTPS scan in August 2016 to check if servers validated received client key exchange values, offering generators of subgroups of order 1, 2 and 7. Our baseline DHE support number counts hosts willing to negotiate a DHE key exchange, and in the case of g_7, if $p-1$ is divisible by 7. We count hosts as “Accepted” if they reply to the `ClientKeyExchange` message with a `Finished` message. For g_7, we expect this to happen with probability $1/7$, suggesting that nearly all of the hosts in our scan did not validate subgroup order.

Group			*Host Counts*			
Source	**Prime Size**	**Subgroup Size**	**HTTPS**	**SMTP**	**POP3S**	**SSH**
RFC 5114 Group 22	1024	160	1,173,147	145	86	0
Amazon Load Balancer	1024	160	277,858	0	1	0
JDK	768	160	146,491	671	16,515	0
JDK	1024	160	52,726	2,445	9,510	0
RFC 5114 Group 24	2048	256	3,543	5	0	6
JDK	2048	224	982	12	20	0
Epson Device	1024	< 948	372	0	0	0
RFC 5114 Group 23	2048	224	371	1,140,363	2	0
Mistyped OpenSSL 512	512	497	0	717	0	0
Other Non-Safe Primes	—	—	6,366	41,964	151	275
Safe Primes	—	—	9,165,709	1,850,086	3,345,331	10,730,246
Total			10,827,565	3,036,408	3,371,616	10,730,527

Table 5.5: **IPv4 top non-safe primes** — Nine non-safe primes account for the majority of hosts using non-safe primes.

divisible by 7, implying that approximately 96% of these hosts fail to validate subgroup order six months after OpenSSL pushed a patch adding group order validation for correctly configured groups.

Second, we measured how many hosts performed even the most basic validation of key exchange values. We attempted to connect to HTTPS hosts with the client key exchange values of $y_c = 0 \bmod p, 1 \bmod p, -1 \bmod p$. As Table 5.4 shows, we found that over 5% of hosts that accepted DHE ciphersuites accepted the key exchange value of $-1 \bmod p$ and derived the `PreMasterSecret` from it. These implementations are vulnerable to a trivial version of the small subgroup confinement attacks described in Section 5.2.1, for *any* prime modulus p. By examining the default web pages of many of these hosts, we identified products from several notable companies including Microsoft, Cisco, and VMWare. When we disclosed these findings, VMWare notified us that they had already applied the fix in the latest version of their products; Microsoft acknowledged the missing checks but chose not to include them since they only use safe primes, and adding the checks may break functionality for some clients that were sending unusual key exchange values; and Cisco informed us that they would investigate the issue.

Of 40.6 M total HTTPS hosts found in our scans, 10.8 M (27%) supported ephemeral Diffie-Hellman, of which 1.6 M (4%) used a non-safe prime, and 309 K (0.8%) used a non-safe prime and reused exponents across multiple connections, making them likely candidates for a small subgroup key recovery attack. We note that the numbers for hosts reusing exponents are an underestimate, since we only mark hosts as such if we found them using the same public Diffie-Hellman value across multiple connections, and some load balancers that cycle among multiple values might have evaded detection.

While 77% of POP3S hosts and 39% of SMTP servers used a non-safe prime, a much smaller number used a non-safe prime and reused exponents (¡0.01% in both protocols), suggesting that the popular implementations (Postfix and Dovecot [46]) that use these primes follow recommendations to use ephemeral Diffie-Hellman values with DSA primes.

Table 5.5 shows nine groups that accounted for the majority of non-safe primes used by hosts in the wild. Over 1.17 M hosts across all of our HTTPS scans negotiated Group 22 in a key exchange. To get a better picture of which implementations provide support for this group, we examined the default web pages of these hosts to identify companies and products, which we show in Table 5.6.

Of the the 307 K HTTPS hosts that both use non-safe primes and reuse exponents, 277 K (90%) belong to hosts behind Amazon's Elastic Load Balancer [9]. These hosts use a 1024-bit prime with a 160-bit subgroup. We set up our own load balancer instance and found that the implementation failed to validate subgroup order. We were able to use a small-subgroup key recovery attack to compute 17 bits of our load balancer's private Diffie-Hellman exponent x_s in only 3813 queries. We responsibly disclosed this vulnerability to Amazon. Amazon informed us that they have removed Diffie-Hellman from their recommended ELB security policy, and are encouraging customers to use the latest policy. In May 2016, we performed additional scans and found that 88% of hosts using this prime no longer repeated exponents. We give a partial factorization for $p-1$ in Table 5.12; the next largest subgroups have 61 and 89 bits and an offline attack against the remaining bits of a 160-bit exponent would take 2^{71} time. For more details on the computation, see Section 5.5.

SSLeay [51], a predecessor for OpenSSL, includes several default Diffie-Hellman primes, including a 512-bit prime. We found that 717 SMTP servers used a version of the OpenSSL 512-bit prime with a single character difference in the hexadecimal representation. The resulting modulus that these servers use for their Diffie-Hellman key exchange is no longer prime. We include the factorization of this modulus along with the factors of the resulting group order in Table 5.12. The use of a composite modulus further decreases the work required to perform a small subgroup attack.

Although TLS also includes static Diffie-Hellman cipher suites that require a DSS certificate, we did not include them in our study; no browser supports static Diffie-Hellman [74], and Censys shows no hosts with DSS certificates, with only 652 total hosts with non-RSA or ECDSA certificates.

5.3 IPsec

IPsec is a set of Layer-3 protocols which add confidentiality, data protection, sender authentication, and access control to IP traffic. IPsec is commonly used to implement VPNs. IPsec uses the Internet Key Exchange (IKE) protocol to determine the keys used to secure a session. IPsec may use IKEv1 [70] or IKEv2 [88]. While IKEv2 is not backwards-compatible with IKEv1, the two protocols are similar in message structure and purpose. Both versions use Diffie-Hellman to negotiate shared secrets. The groups used are limited to a fixed set of pre-determined choices, which include the DSA groups from RFC 5114, each assigned a number by IANA [88,89,96].

IKEv1 IKEv1 [70,100,113] has two basic methods for authenticated key exchange: Main Mode and Aggressive Mode. Main Mode requires six messages to establish the requisite state. The initiator sends a Security Association (SA) payload, containing a selection of cipher suites and Diffie-Hellman groups they are willing to negotiate. The responder selects a cipher and responds with its own SA payload. After the cipher suite is selected, the initiator and responder both transmit Key Exchange (KE) payloads containing public Diffie-Hellman values for the chosen group. At this point, both parties compute shared key materials, denoted SKEYID. When using signatures for authentication, SKEYID is computed $\texttt{SKEYID} = \mathrm{prf}(N_i|N_r, g^{x_i x_r})$. For the other two authentication modes, pre-shared key and public-key encryption, SKEYID is derived from the pre-shared key and session cookies, respectively, and does not depend on the negotiated Diffie-Hellman shared secret.

Each party then in turn sends an authentication message (AUTH) derived from a hash over SKEYID and the handshake. The authentication messages are encrypted and authenticated using keys derived from the Diffie-Hellman secret $g^{x_i x_r}$. The responder only sends her AUTH message after receiving and validating the initiator's AUTH message.

Aggressive Mode operates identically to Main Mode, but in order to reduce latency, the initiator sends SA and KE messages together, and the responder replies with its SA, KE, and AUTH messages together. In aggressive mode, the responder sends an authentication message first, and the authentication messages are not encrypted.

IKEv2 IKEv2 [87, 88] combines the SA and KE messages into a single message. The initiator provides a best guess ciphersuite for the KE message. If the responder accepts that proposal and chooses not to renegotiate, the responder replies with a single message containing both SA and KE payloads. Both parties then send and verify AUTH messages, starting with the initiator. The authentication messages are encrypted using session keys derived from the SKEYSEED value which is derived from the negotiated Diffie-Hellman shared secret. The standard authentication modes use public-key signatures over the handshake values.

5.3.1 Small Subgroup Attacks in IPsec

There are several variants of small subgroup attacks against IKEv1 and IKEv2. We describe the attacks against these protocols together in this section.

Small subgroup confinement attacks First, consider attacks that can be carried out by an attacking initiator or responder. In IKEv1 Main Mode and in IKEv2, either peer can carry out a small subgroup confinement attack against the other by sending a generator of a small subgroup as its key exchange value. The attacking peer must then guess the other peer's view of the Diffie-Hellman shared secret to compute the session keys to encrypt its authentication message, leading to a mostly online attack. However, in IKEv1 Aggressive Mode, the responder sends its AUTH message before the initiator, and this value is not encrypted with a session key. If signature authentication is being used, the SKEYID and resulting hashes are derived from the Diffie-Hellman shared secret, so the initiator can perform an offline brute-force attack against the responder's authentication message to learn their exponent in the small subgroup.

Now, consider a man-in-the-middle attacker. Bhargavan, Delignat-Lavaud, and Pironti [29] describe a transcript synchronization attack against IKEv2 that relies on a small subgroup confinement attack. A man-in-the-middle attacker initiates simultaneous connections with an initiator and a responder using identical nonces, and sends a generator g_i for a subgroup of small order q_i to each as its KE message. The two sides have a $1/q_i$ chance of negotiating an identical shared secret, so an authentication method depending only on nonces and shared secrets could be forwarded, and the session keys would be identical.

If the attacker also has knowledge of the secrets used for authentication, more attacks are possible. Similar to the attack described for TLS, such an attacker can use a small subgroup confinement attack to force a connection to use weak encryption. The attacker only needs to rewrite a small number of handshake messages; any further encrypted communications can then be decrypted at leisure without requiring the man-in-the-middle attacker to continuously rewrite the connection. We consider a man-in-the-middle attacker who modifies the key exchange message from both the initiator and the responder to substitute a generator g_i of a subgroup of small order q_i. The attacker must then replace the handshake authentication messages, which would require knowledge of the long-term authentication secret. We describe this attack for each of pre-shared key, signatures, and public-key authentication.

For pre-shared key authentication in IKEv1 Main Mode, IKEv1 Aggressive Mode, and IKEv2, the man-in-the-middle attacker must only know the pre-shared key to construct the authentication hash; the authentication message does not depend on the negotiated Diffie-Hellman shared secret. With probability $1/q_i$, the two parties will agree on the Diffie-Hellman shared secret. The attacker can then brute force this value after viewing messages encrypted with keys derived from it.

For signature authentication in IKEv1 Main Mode and in IKEv2, the signed hash transmitted from each side is derived from the nonces and the negotiated shared secret, which is confined to one of q_i possible values. The attacker must know the private signing keys for both initiator and responder and brute force SKEYID from the received signature in order to forge the modified authentication signatures on each side. The communicating parties will have a q_i chance of agreeing on the same value for the shared secret to allow the attack to succeed. For IKEv1 Aggressive Mode, the attack can be made to succeed every time. The responder's key exchange message is sent together with their signature which depends on the negotiated shared secret, so the man-in-the-middle attacker can brute force the q_i possible values of the responders private key x_r and replace the responder's key exchange message with $q_i^{x_r}$, forging an appropriate signature with their knowledge of the signing key.

For public key authentication in IKEv1 Main Mode, IKEv1 Aggressive Mode, and IKEv2, the attacker must know the private keys corresponding to the public keys used to encrypt the ID and nonce values on both sides in order to forge a valid authentication hash. Since the authentication does not depend on the shared Diffie-Hellman negotiated value, a man-in-the-middle attacker must then brute force the negotiated shared key once they receives a message encrypted with the derived key. The two parties will agree on their view of the shared key with probability $1/q_i$, allowing the attack to succeed.

Small subgroup key recovery attacks Similar to TLS, an IKE responder that reuses private exponents and does not verify that the initiator key exchange values are in the correct subgroup is vulnerable to a small subgroup key recovery attack. The most recent version of the IKEv2 specification has a section discussing reuse of Diffie-Hellman exponents, and states that "because computing Diffie-Hellman exponentials is computationally expensive, an endpoint may find it advantageous to reuse those exponentials for multiple connection setups" [88]. Following this recommendation could leave a host open to a key recovery attack, depending on how exponent reuse is implemented. A small subgroup key recovery attack on IKE would be primarily offline for IKEv1 with signature authentication and for IKEv2 against the initiator.

For each subgroup of order q_i, the attacker's goal is to obtain a responder `AUTH` message, which depends on the secret chosen by the responder. If an `AUTH` message can be obtained, the attacker can brute-force the responder's secret within the subgroup offline. This is possible if the server supports IKEv1 Aggressive Mode, since the server authenticates before the client, and signature authentication produces a value dependent on the negotiated secret. In all other IKE modes, the client authenticates first, leading to an online attack. The flow of the attack is identical to TLS; for more details see Section 5.2.

Ferguson and Schneier [52] describe a hypothetical small-subgroup attack against the initiator where a man-in-the-middle attacker abuses undefined behavior with respect to UDP packet retransmissions. A malicious party could "retransmit" many key exchange messages to an initiator and potentially receive a different authentication message in response to each, allowing a mostly offline key recovery attack.

5.3.2 Implementations

We examined several open-source IKE implementations to understand server behavior. In particular, we looked for implementations that generate small Diffie-Hellman exponents, repeat exponents across multiple connections, or do not correctly validate subgroup order. Despite the suggestion in IKEv2 RFC 7296 to reuse exponents [88], none of the implementations that we examined reused secret exponents.

All implementations we reviewed are based on FreeS/WAN [12], a reference implementation of IPSec. The final release of FreeS/Wan, version 2.06, was released in 2004. Version 2.04 was forked into Openswan [15] and strongSwan [16], with a further fork of Openswan into Libreswan [13] in 2012. The final release of FreeS/WAN used constant length 256-bit exponents but did not support RFC 5114 DSA groups, offering only the Oakley 1024-bit and 1536-bit groups that use safe primes.

Openswan does not generate keys with short exponents. By default, RFC 5114 groups are not supported, although there is a compile-time option that can be explicitly set to enable support for DSA groups. strongSwan both supports RFC 5114 groups and has explicit hard-coded exponent sizes for each group. The exponent size for each of the RFC 5114 DSA groups matches the subgroup size. However, these exponent sizes are only used if the `dh_exponent_ansi_x9_42` configuration option is set. It also includes a routine inside an `#ifdef` that validates subgroup order by checking that $g^q \equiv 1 \bmod p$, but validation is not enabled by default. Libreswan uses Mozilla Network Security Services (NSS) [8] to generate Diffie-Hellman keys. As discussed in Section 5.2.3, NSS generates short exponents for

Diffie-Hellman groups. Libreswan was forked from Openswan after support for RFC 5114 was added, and retains support for those groups if it is configured to use them.

Although none of the implementations we examined were configured to reuse Diffie-Hellman exponents across connections, the failure to validate subgroup orders even for the pre-specified groups renders these implementations fragile to future changes and vulnerable to subgroup confinement attacks.

Several closed source implementations also provide support for RFC 5114 Group 24. These include Cisco's IOS [38], Juniper's Junos [83], and Windows Server 2012 R2 [106]. We were unable to examine the source code for these implementations to determine whether or not they validate subgroup order.

5.3.3 Measurements

We performed a series of Internet scans using ZMap to identify IKE responders. In our analysis, we only consider hosts that respond to our ZMap scan probes. Many IKE hosts that filter their connections based on IP are excluded from our results. We further note that, depending on VPN server configurations, some responders may continue with a negotiation that uses weak parameters until they are able to identify a configuration for the connecting initiator. At that point, they might reject the connection. As an unauthenticated initiator, we have no way of distinguishing this behavior from the behaviour of a VPN server that legitimately accepts weak parameters. For a more detailed explanation of possible IKE responder behaviors in response to scanning probes, see Wouters [134].

In October 2016, we performed a series of scans offering the most common cipher suites and group parameters we found in implementations to establish a baseline population for IKEv1 and IKEv2 responses. For IKEv1, the baseline scan offered Oakley groups 2 and 14 and RFC 5114 groups 22, 23, and 24 for the group parameters; SHA1 or SHA256 for the hash function; pre-shared key or RSA signatures for the authentication method; and AES-CBC, 3DES, and DES for the encryption algorithm. Our IKEv2 baseline scan was similar, but also offered the 256-bit and 384-bit ECP groups and AES-GCM for authenticated encryption.

On top of the baseline scans, we performed additional scans to measure support for the non-safe RFC 5114 groups and for key exchange parameter validation. Table 5.7 shows the results of the October IKE scans. For each RFC 5114 DSA group, we performed four handshakes with each host; the first tested for support by sending a valid client key exchange value, and the three others tested values that should be rejected by a properly-validating host. We did not scan using the key exchange value 0 because of a vulnerability present in unpatched Libreswan and Openswan implementations that causes the IKE daemon to restart when it receives such a value [5].

We considered a host to accept our key exchange value if after receiving the value, it continued the handshake without any indication of an error. We found that 33.2% of IKEv1 hosts and 17.7% of IKEv2 hosts that responded to our baseline scans supported using one of the RFC 5114 groups, and that a surprising number of hosts failed to validate key exchange values. 24.8% of IKEv1 hosts that accepted Group 23 with a valid key exchange value also accepted 1 mod p or -1 mod p as a key exchange value, even though this is explicitly warned against in the RFC [112]. This behavior leaves these hosts open to a small subgroup confinement attack even for safe primes, as described in Section 5.1.6.

For safe groups, a check that the key exchange value is strictly between 1 and $p-1$ is sufficient validation. However, when using non-safe DSA primes, it is also necessary to verify that the key exchange value lies within the correct subgroup (i.e., $y^q \equiv 1 \bmod p$). To test this case, we constructed a generator of a subgroup that was not the intended DSA subgroup, and offered that as our key exchange value. We did not find any IKEv1 hosts that rejected this key exchange value after previously accepting a valid key exchange value for the given group. For IKEv2, the results were similar with the exception of Group 24, where still over 93% of hosts accepted this key exchange value. This suggests that almost no hosts supporting DSA groups are correctly validating subgroup order.

We observed that across all of the IKE scans, 109 IKEv1 hosts and 52 IKEv2 hosts repeated a key exchange value. This may be due to entropy issues in key generation rather than static Diffie-Hellman exponents; we also found 15,891 repeated key exchange values across different IP addresses. We found no hosts that used both repeated key exchange values and non-safe groups. We summarize these results in Table 5.3.

5.4 SSH

SSH contains three key agreement methods that make use of Diffie-Hellman. The “Group 1” and “Group 14” methods denote Oakley Group 2 and Oakley Group 14, respectively [135]. Both of these groups use safe primes. The third method, “Group Exchange”, allows server to select a custom group [55]. The group exchange RFC specifies that all custom groups should use safe primes. Despite this, RFC 5114 notes that group exchange method allows for its DSA groups in SSH, and advocates for their immediate inclusion [96].

In all Diffie-Hellman key agreement methods, after negotiating cipher selection and group parameters, the SSH client generates a public key exchange value $y_c = g^{x_c} \bmod p$ and sends it to the server. The server computes its own Diffie-Hellman public value $y_s = g^{x_s} \bmod p$ and sends it to the client, along with a signature from its host key over the resulting shared secret $Y = g^{x_s x_c} \bmod p$ and the hash of the handshake so far. The client verifies the signature before continuing.

5.4.1 Small Subgroup Attacks in SSH

Small subgroup confinement attacks An SSH client could execute a small subgroup confinement attack against an SSH server by sending a generator g_i for a subgroup of small order q_i as its client key exchange, and immediately receive the server's key exchange $g^{x_s} \bmod p$ together with a signature that depends on the server's view of the shared secret $Y_s = g_i^{x_s} \bmod p$. For small q_i, this allows the client to brute force the value of $x_s \bmod q_i$ offline and compare to the server's signed handshake to learn the correct value of $x_s \bmod q_i$. To avoid this, the SSH RFC specifically recommends using safe primes, and to use exponents at least twice the length of key material derived from the shared secret [55].

If client and server support Diffie-Hellman group exchange and the server uses a non-safe prime, a man in the middle with knowledge of the server's long-term private signing key can use a small subgroup confinement attack to man-in-the-middle the connection without having to rewrite every message. The attack is similar to the case of TLS: the man in the middle modifies the server group and key exchange messages, leaving the prime unchanged, but substituting a generator g_i of a subgroup of small order q_i for the group generator and g_i for the server's key exchange value y_s. The client then responds with a client key exchange message $y_c = g_i^{x_c} \bmod p$, which the man in the middle leaves unchanged. The attacker then forges a correct signature for the modified server group and key exchange messages and passes it to the client. The server's view of the shared secret is $g_i^{x_c x_s} \bmod p$, and the client's view of the shared secret is $g_i^{x_c} \bmod p$. As in the attack described for TLS, these views are identical when $x_s \equiv 1 \bmod q_i$, so this connection will succeed with probability $1/q_i$. For a small enough q_i, this enables a man in the middle to use a compromised server signing key to decrypt traffic with a reasonable probability of success, while only requiring tampering with the initial handshake messages, rather than having to actively rewrite the entire connection for the duration of the session.

Small subgroup key recovery attacks Since the server immediately sends a signature over the public values and the Diffie-Hellman shared secret, an implementation using static exponents and non-safe primes that is vulnerable to a small subgroup confinement attack would also be vulnerable to a mostly offline key recovery attack, as a malicious client would only need to send a single key exchange message per subgroup.

5.4.2 Implementations

Censys [45] SSH banner scans show that the two most common SSH server implementations are Dropbear and OpenSSH. Dropbear group exchange uses hard-coded safe prime parameters from the Oakley groups and validates that client key exchange values are greater than 1 and less than $p-1$. While OpenSSH only includes safe primes by default, it does provide the ability to add additional primes and does not provide the ability to specify subgroup orders. Both OpenSSH and Dropbear generate fresh exponents per connection.

We find one SSH implementation, Cerberus SFTP server (FTP over SSH), repeating server exponents across connections. Cerberus uses OpenSSL, but fails to set `SSL_OP_SINGLE_DH_USE`, which was required to avoid exponent reuse prior to OpenSSL 1.0.2f.

5.4.3 Measurements

Of the 15.2 M SSH servers on Censys, of which 10.7 M support Diffie-Hellman group exchange, we found that 281 used a non-safe prime, and that 1.1 K reused Diffie-Hellman exponents. All but 26 of the hosts that reused exponents had banners identifying the Cerberus SFTP server. We encountered no servers that both reused exponents and used non-safe primes.

We performed a scan of 1% of SSH hosts in February 2016 offering the key exchange values of $y_c = 0 \bmod p, 1 \bmod p$ and $p-1 \bmod p$. As Table 5.8 shows, 33% of SSH hosts failed to validate group order when we sent the key exchange value $p-1 \bmod p$. Even when safe groups are used, this behaviour allows an attacker to learn a single bit of the private exponent, violating the decisional Diffie-Hellman assumption and leaving the implementation open to a small subgroup confinement attack (Section 5.2.1).

5.5 Factoring Group Orders of Non-Safe Primes

Across all scans, we collected 41,847 unique groups with non-safe primes. To measure the extent to which each group would facilitate a small subgroup attack in a vulnerable implementation, we attempted to factor $(p-1)/2$. We used the GMP-ECM [137] implementation of the elliptic curve method for integer factorization on a local cluster with 288 cores over a several-week period to opportunistically find small factors of the group order for each of the primes.

Given a group with prime p and a generator g, we can check whether the generator generates the entire group or generates a subgroup by testing whether $g^{q_i} \equiv 1 \bmod p$ for each factor q_i of $(p-1)/2$. When $g^{q_i} \equiv 1 \bmod p$, then if q_i is prime, we know that q_i is the exact order of the subgroup generated by g; otherwise q_i is a multiple of the order of the subgroup. We show the distribution of group order for groups using non-safe primes in Table 5.9. We were able to completely factor $p-1$ for 4,701 primes. For the remaining primes, we did not obtain enough factors of $(p-1)/2$ to determine the group order.

Of the groups where we were able to deduce the exact subgroup orders, several thousand had a generator for a subgroup that was either 8, 32, or 64 bits shorter than the prime itself. Most of these were generated by the Xlight FTP server, a closed-source implementation supporting SFTP. It is not clear whether this behavior is intentional or a bug in an implementation intending to generate safe primes. Primes of this form would lead to a more limited subgroup confinement or key recovery attack.

Given the factorization of $(p-1)/2$, and a limit for the amount of online and offline work an attacker is willing to invest, we can estimate the vulnerability of a given group to a hypothetical small subgroup key recovery attack. For each subgroup of order q_i, where q_i is less than the online work limit, we can learn q_i bits of the secret key via an online brute-force attack over all elements of the subgroup. To recover the remaining bits of the secret key, an attacker could use the Pollard lambda algorithm, which runs in time proportional to the square root of the remaining search space. If this runtime is less than the offline work limit, we can recover the entire secret key. We give work estimates for the primes we were able to factor and the number of hosts that would be affected by such a hypothetical attack in Table 5.10.

The DSA groups introduced in RFC 5114 [96] are of particular interest. We were able to completely factor $(p-1)/2$ for both Group 22 and Group 24, and found several factors for Group 23. We give these factorizations in Table 5.12. In Table 5.11, we show the amount of online and offline work required to recover a secret exponent for each of the RFC 5114 groups. In particular, an exponent of the recommended size used with Group 23 is fully recoverable via a small subgroup attack with 33 bits of online work and 47 bits of offline work.

5.6 Discussion

The small subgroup attacks require a number of special conditions to go wrong in order to be feasible. For the case of small subgroup confinement attacks, a server must both use a non-safe group and fail to validate subgroup order; the widespread failure of implementations to implement or enable group order validation means that large numbers of hosts using non-“safe” primes are vulnerable to this type of attack.

For a full key recovery attack to be possible the server must additionally reuse a small static exponent. In one sense, it is surprising that any implementations might satisfy all of the requirements for a full key recovery attack at once. However, when considering all of the choices that cryptographic libraries leave to application developers when using Diffie-Hellman, it is surprising that any protocol implementations manage to use Diffie-Hellman securely at all.

We now use our results to draw lessons for the security and cryptographic communities, provide recommendations for future cryptographic protocols, and suggest further research.

RFC 5114 Design Rationale Neither NIST SP 800-56A nor RFC 5114 give a technical justification for fixing a much smaller subgroup order than the prime size. Using a shorter private exponent comes with performance benefits. However, there are no known attacks that would render a short exponent used with a safe prime less secure than an equivalently-sized exponent used with in a subgroup with order matched to the exponent length. The cryptanalyses of both short exponents and small subgroups are decades old.

If anything, the need to perform an additional modular exponentiation to validate subgroup order makes Diffie-Hellman over DSA groups *more* expensive than the safe prime case, for identical exponent lengths. As a more minor effect, a number field sieve-based cryptanalytic attack against a DSA prime is computationally slightly easier than against a safe prime. The design rationale may have its roots in preferring to implicitly use the assumption that Diffie-Hellman is secure for a small prime-order subgroup without conditions on exponent length, rather than assuming Diffie-Hellman with short exponents is secure inside a group of much larger order. Alternatively, this insistence may stem from the fact that the security of DSA digital signatures requires the secret exponent to be uniformly random, although no such analogous attacks are known for Diffie-Hellman key exchange. [108] Unfortunately, our empirical results show that the necessity to specify and validate subgroup order for Diffie-Hellman key exchange makes implementations more fragile in practice.

Cryptographic API design Most cryptographic libraries are designed with a large number of potential options and knobs to be tuned, leaving too many security-critical choices to the developers, who may struggle to remain current with the diverse and ever-increasing array of cryptographic attacks. These exposed knobs are likely due to a prioritization of performance over security. These confusing options in cryptographic implementations are not confined to primitive design: Georgiev et al. [60] discovered that SSL certificate validation was broken in a large number of non-browser TLS applications due to developers misunderstanding and misusing library calls. In the case of the small subgroup attacks, activating most of the conditions required for the attack will provide slight performance gains for an application: using a small exponent decreases the work required for exponentiation, reusing Diffie-Hellman exponents saves time in key generation, and failing to validate subgroup order saves another exponentiation. It is not reasonable to assume that applications developers have enough understanding of algebraic groups to be able to make the appropriate choices to optimize performance while still providing sufficient security for their implementation.

Cryptographic standards Cryptographic recommendations from standards committees are often too weak or vague, and, if strayed from, provide little recourse. The purpose of standardized groups and standardized validation procedures is to help remove the onus from application developers to know and understand the details of the cryptographic attacks. A developer should not have to understand the inner workings of Pollard lambda and the number field sieve in order to size an exponent; this should be clearly and unambiguously defined in a standard. However, the tangle of RFCs and standards attempting to define current best practices in key generation and parameter sizing do not paint a clear picture, and instead describe complex combinations of approaches and parameters, exposing the fragility of the cryptographic ecosystem. As a result, developers often forget or ignore edge cases, leaving many implementations of Diffie-Hellman too close to vulnerable for comfort. Rather than provide the bare minimums for security, the cryptographic recommendations from standards bodies should be designed for defense-in-depth such that a single mistake on the part of a developer does not have disastrous consequences for security. The principle of defense-in-depth has been a staple of the systems security community; cryptographic standards should similarly be designed to avoid fragility.

Protocol design The interactions between cryptographic primitives and the needs of protocol designs can be complex. The after-the-fact introduction of RFC 5114 primes illustrates some of the unexpected difficulties: both IKE and SSH specified group validation only for safe primes, and a further RFC specifying extra group validation checks needed to be defined for IKE. Designing protocols to encompass many unnecessary functions, options, and extensions leaves room for implementation errors and makes security analysis burdensome. IKE is a notorious example of a difficult-to-implement protocol with many edge cases. Just Fast Keying (JFK), a protocol created as a successor to IKEv1, was designed to be an exceedingly simple key exchange protocol without the unnecessarily complicated negotiations present in IKE [19]. However, the IETF instead standardized IKEv2, which is nearly as complicated as IKEv1. Protocols and cryptosystems should be designed with the developer in mind—easy to implement and verify, with limited edge cases. The worst possible outcome is a system that appears to work, but provides less security than expected.

To construct such cryptosystems, secure-by-default primitives are key. As we show in this paper, finite-field based Diffie-Hellman has many edge cases that make its correct use difficult, and which occasionally arise as bugs at the protocol level. For example, SSH and TLS allow the server to generate arbitrary group parameters and send them to the client, but provide no mechanism for the server to specify the group order so that the client can validate the parameters. Diffie-Hellman key exchange over groups with different properties cannot be treated as a black-box primitive at the protocol level.

Recommendations As a concrete recommendation, modern Diffie-Hellman implementations should prefer elliptic curve groups over safe curves with proper point validation [26]. These groups are much more efficient and have shorter key sizes than finite-field Diffie-Hellman at equivalent security levels. The TLS 1.3 draft includes a list of named curves designed to modern security standards [118]. If elliptic curve Diffie-Hellman is not an option, then implementations should follow the guidelines outlined in RFC 7919 for selecting finite field Diffie-Hellman primes [61]. Specifically, implementations should prefer "safe" primes of documented provenance of at least 2048 bits, validate that key exchange values are strictly between 1 and $p-1$, use ephemeral key exchange values for every connection, and use exponents of at least 224 bits.

Company	**Product(s)**	**Count**
Ubiquiti Networks	airOS/EdgeOS	272,690
Cisco	DPC3848VM Gateway	65,026
WatchGuard	Fireware XTM	62,682
Supermicro	IPMI	42,973
ASUS	AiCloud	39,749
Electric Sheep Fencing	pfSense	14,218
Bouygues Telecom	Bbox	13,387
Other	—	135,432

Table 5.6: **HTTPS support for RFC5114 Group 22** — In a 100% HTTPS scan performed in October 2016, we found that of the 12,835,911 hosts that accepted Diffie-Hellman key exchange, 901,656 used Group 22. We were able to download default web pages for 646,157 of these hosts, which we examined to identify companies and products.

			Client key exchange public values offered…		
Protocol	**Groups Offered**	**Support**	1 mod p	-1 mod p	g_s mod p
IKEv1	Group 22	332.4 K	82.6 K	78.5 K	332.4 K
	Group 23	333.4 K	82.5 K	82.5 K	333.4 K
	Group 24	379.8 K	93.9 K	95.2 K	379.8 K
	Baseline (Groups 2, 14, 22, 23, 24)	1139.3 K	–	–	–
IKEv2	Group 22	182.1 K	553	553	181.9 K
	Group 23	181.9 K	542	550	180.1 K
	Group 24	213.0 K	2245	2173	200.0 K
	Baseline (Groups 2, 14, 19, 20, 22, 23, 24)	1203.7 K	–	–	–

Table 5.7: **IKE group support and validation** — We measured support for RFC5114 DSA groups in IKEv1 and IKEv2 and test for key exchange validation by performing a series of 100% IPv4 scans in October 2016. For Group 23, g_s is a generator of a subgroup with order 3, and for Groups 22 and 24, g_s is a generator of a subgroup of order 7.

Key Exchange Value	**Handshake Initiated**	**Accepted**
0 mod p	175.6 K	5.7 K
1 mod p	175.0 K	43.9 K
-1 mod p	176.0 K	59.0 K

Table 5.8: **SSH validation** — In a 1% SSH scan performed in February 2016, we sent the key exchange values $y_c = 0, 1$ and $p - 1$. We count hosts as having initiated a handshake if they send a `SSH_MSG_KEX_DH_GEX_GROUP`, and we count hosts as "Accepted" if they reply to the client key exchange message with a `SSH_MSG_KEX_DH_GEX_REPLY`.

Prime	**Exact Order Known**							**Exact Order Unknown**	
lg(p)	160 bits	224 bits	256 bits	300 bits	lg$(p)-8$	lg$(p)-32$	lg$(p)-64$	Unlikely DSA	Likely DS
512	3	0	0	0	5	0	0	760	4
768	4	0	0	4	2,685	0	0	220	1,40
1024	29	0	0	0	323	944	176	1,559	26,88
2048	0	1	1	0	0	0	0	1,128	4,89
3072	0	0	0	0	0	5	0	9	15
4096	4	0	0	0	0	0	0	20	18
8192	0	0	0	0	0	0	0	0	
Other	0	0	0	0	0	0	0	400	1

Table 5.9: **Distribution of orders for groups with non-safe primes** — For groups for which we were able to determine the subgroup order exactly, 160-bits subgroup orders are common. We classify other groups to be likely DSA groups if we know that the subgroup order is at least 8 bits smaller than the prime.

	Work (bits)		***HTTPS***		***MAIL***		***SSH***	
Exponent	**Online**	**Offline**	**Groups**	**Hosts**	**Groups**	**Hosts**	**Groups**	**Hosts**
160	20	30	3	2	3	7	0	0
160	30	45	517	1,996	1963	1,143,524	11	10
160	40	60	3,701	8,495	13,547	1,159,853	109	68
224	20	30	0	0	0	0	0	0
224	30	45	2	2	14	16	0	0
224	40	60	307	691	1039	1,141,840	3	1
256	20	30	0	0	0	0	0	0
256	30	45	0	0	1	1	0	0
256	40	60	42	478	180	1,140,668	0	0

Table 5.10: **Full key recovery attack complexity** — We estimate the amount of work required to carry out a small subgroup key recovery attack, and show the prevalence of those groups in the wild. Hosts are vulnerable if they reuse exponents and fail to check subgroup order.

Group	**Exponent Size**	**Online Work**	**Offline Work**
Group 22	160	8	72
Group 23	224	33	47
Group 24	256	32	94

Table 5.11: **Attacking RFC 5114 groups** — We show the log of the amount of work in bits required to perform a small subgroup key recovery attack against a server that both uses a static Diffie-Hellman exponent of the same size as the subgroup order and fails to check group order.

Source	Factored Completely?	Order Factorization

RFC 5114 Group 22	Yes	`2^3 * 7 * df * 183a872bdc5f7a7e88170937189 * 228c5a311384c02e1f28`
		`7d66c65a60728c353e32ece8be1 * f518aa8781a8df278aba4e7d64b7cb9d494`
		`d6a69682661ca6e590b447e66ebd1bbdeab5e6f3744f06f46cf2a8300622ed500`
		`a53d30113995663a447dcb8e81bc24d988edc41f21`

RFC 5114 Group 23	No	`3^2 * 5 * 2b * 49 * 9d * 5e9a5 * 93ee1 * 2c3f0539 * 136c58359 * 1`
		`a378eb0d * 801c0d34c58d93fe997177101f80535a4738cebcbf389a99b36371`
		`f6fc6dc24fef3f56e1c216523b3210d27b6c078b32b842aa48d35f230324e48f6`
		`82843a78f264495542be4a95cb05e41f80b013f8b0e3ea26b84cd497b43cc9326`
		`f8ea3cc84139f0667100d426b60b9ab82b8de865b0cbd633f4136662201100663`
		`066efe4ab4f1b2e99d96adfaf1721447b167cb49c372efcb82923b3731433cecb`
		`41b5d11fb3328851084f74de823b5402f6b038172348a147b1ceac47722e31a72`

RFC 5114 Group 24	Yes	`7 * d * 9f5 * 22acf * bd9f34b1 * 8cf83642a709a097b447997640129da2`
		`ba308b0fe64f5fbd3 * 15adfe949ebb242e5cd0978fac1b43fdbd2e5b0c5f489`
		`b20596d98ad0a9e3fd98876413d926f41a8b918d2ec4b018a30efe5e336bf3c7c`
		`acf3bb389f68ad0c4ed2f0b1dbb970293741eb6509c64e731802259a639a7f57d`
		`bcdbdc50555b76d9c335c1fa4e11a8351f1bf4730dd67ffed877cc13e8ea40c7d`
		`ef1159eca75a2359f5e0284cd7f3b982c32e5c51dbf51b45f4603ef46bae52873`
		`fcf3b44fe3da5999daadf5606eb828fc57e46561be8c6a866361`

Amazon Load Balancer	No	`2 * 3 * 5 * edb * 181ac5dbfe5ce13b * 18aa349859e9e9de09b7d65 * 94`
		`2f6cb2dbc22eb1fc21d4929 * 2de9f1171a2493d46a31d508b63532cdf86d21d`
		`b0b722856a504ed4916e0484fe4ba5f5f4a9fff28a1233b728b3d043aec37c4f1`
		`1e93cb52be527395e45db487b61daadded9c8ec35`

Mistyped OpenSSL 512 “Prime” Factors	Yes	`5 * b * a9b461e1636f4b51ef * 1851583cf5f9f731364e4aa6cdc2cac4f01*`
		`df4baf46c7fa7d1f4dfe184f9d22848325a91c519f79023a4526d8369e86b`

Mistyped OpenSSL 512 Order Factors	Yes	`2^13 * 3^3 * 5^2 * 11^2 * 269 * 295 * 4d5 * 97c3 * 9acfe7 * 8cdd0`
		`b564eecd613536818f949 * 146d410923e999f8c291048dc6feffcebf8b9e99e`
		`e49b393256c23c9`

Table 5.12: **Group order factorization for common non-safe primes** — We used the elliptic curve method to factor $(p-1)/2$ for each of the non-safe primes we found while scanning, as well as the mistyped OpenSSL “prime”.

CHAPTER 6

Conclusion and Future Work

This is an outline

List of how the empirical methods discussed earlier had impact (pull-out relevant results).

Did any of this research go anywhere beyond that? Yes, it had a clear impact on TLS 1.3.

6.1 TLS 1.3

Point out that the "informing future protocol design" is effectively "making TLS 1.3 better". This is the "improve the security of the Internet in the future" part.

Call out explicit TLS 1.3 design decisions based on results described in this dissertation. Perhaps some of this should be interleaved into the earlier sections?

6.2 Engineering Challenges

For not being a "systems" field, there is still an absurd amount of engineering that largely goes unacknowledged, in order to write good measurement papers. Discuss some of these examples.

Is this a fundamental state of being of the methodology, or are we doing something wrong?

6.3 Broader applicability of empirical methods

Security driven by data. Is any of this relevant to users who are just trying to secure this own networks?

Extending measurement from aggregations and ecosystems to behavior of individual hosts at global scale. Can we track individual hosts appearing and disappearing, and map changes to configuration in real-time?

Should empiricism be part of more traditional cryptography education and research?

Some word of warning about how measurement doesn’t solve all our problems, and how measuring the wrong things makes things worse, e.g. Robert McNamera’s use of body count as a metric during the Vietnam War.

BIBLIOGRAPHY

[1] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.

[2] Finite field cryptography based samples. http://csrc.nist.gov/groups/ST/toolkit/documents/Examples/KS_FFC_All.pdf.

[3] NSS dh.c. https://hg.mozilla.org/projects/nss/file/tip/lib/freebl/dh.c.

[4] Introducing PF_RING ZC. ntop Blog, April 2014. http://www.ntop.org/pf_ring/introducing-pf_ring-zc-zero-copy/.

[5] CVE-2015-3240. MITRE CVE-ID CVE-2015-3240., August 2015. http://cve.mitre.org/cgi-bin/cvename.cgi?name=2015-3240.

[6] Mozilla bug tracker, November 2015. https://bugzilla.mozilla.org/show_bug.cgi?id=1160139.

[7] OpenSSL changes, January 2015. https://www.openssl.org/news/cl102.txt.

[8] Overview of NSS, September 2015. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Overview.

[9] Amazon Elastic Load Balancer, 2016. https://aws.amazon.com/elasticloadbalancing/.

[10] Botan, 2016. https://github.com/randombit/botan.

[11] CVE-2016-0701. MITRE CVE-ID CVE-2016-0701., January 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0701.

[12] FreeS/WAN, 2016. http://www.freeswan.org/.

[13] Libreswan, 2016. https://libreswan.org/.

[14] OpenSSL security advisory [28th Jan 2016], January 2016. https://www.openssl.org/news/secadv/20160128.txt.

[15] Openswan, 2016. https://www.openswan.org/.

[16] strongSwan, 2016. https://www.strongswan.org/.

[17] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[18] David Adrian, Zakir Durumeric, Gulshan Singh, and J. Alex Halderman. Zippier ZMap: Internet-wide scanning at 10Gbps. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[19] William Aiello, Steven M Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D Keromytis, and Omer Reingold. Just fast keying: Key agreement in a hostile internet. *Transactions on Information and System Security (TISSEC)*, 7(2):242–273, 2004.

[20] Nadhem J Al Fardan and Kenneth G Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[21] Nadhem J AlFardan, Daniel J Bernstein, Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of RC4 in TLS. In *22nd USENIX Security Symposium*, 2013.

[22] Ross Anderson and Serge Vaudenay. Minding your p's and q's. In *ASIACRYPT*, 1996.

[23] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In *CRYPTO*, 2012.

[24] Elaine B Barker, Don Johnson, and Miles E Smid. Sp 800-56a. recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised). 2007.

[25] D. J. Bernstein. SYN cookies, 1996. http://cr.yp.to/syncookies.html.

[26] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2014. https://safecurves.cr.yp.to/.

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

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

[29] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. Verified contributive channel bindings for compound authentication. In *Network and Distributed System Security Symposium*, 2015.

[30] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, 2014.

[31] Karthikeyan Bhargavan and Ga"etan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium*, February 2016.

[32] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology (CRYPTO)*, 1998.

[33] Cyril Bouvier, Pierrick Gaudry, Laurent Imbert, Hamza Jeljeli, and Emmanuel Thomé. New record for discrete logarithm in a prime finite field of 180 decimal digits, 2014. http://caramel.loria.fr/p180.txt.

[34] Peter Bowen, December 2015. https://cabforum.org/pipermail/public/2015-December/006507.html.

[35] Wolfgang Breyha, David Durvaux, Tobias Dussa, L. Aaron Kaplan, Florian Mendel, Christian Mock, Manuel Koschuch, Adi Kriegisch, Ulrich Pöschl, Ramin Sabet, Berg San, Ralf Schlatterbeck, Thomas Schreck, Alexander Würstlein, Aaron Zauner, and Pepi Zawodsky. Better crypto: Applied crypto hardening, 2016. https://bettercrypto.org/static/applied-crypto-hardening.pdf.

[36] Ran Canetti and Hugo Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *Crypto*, 2002.

[37] Wan-Teh Chang and Adam Langley. QUIC crypto, 2014. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1.

[38] Cisco. Security for VPNs with IPsec configuration guide, 2016. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/sec_conn_vpnips/configuration/xe-3s/sec-sec-for-vpns-w-ipsec-xe-3s-book.html.

[39] D. Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, January 1994.

[40] CVE-2015-0293. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0293.

[41] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium*, 2015.

[42] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[43] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22(6):644–654, 1976.

[44] Thai Duong and Juliano Rizzo. Here come the xor ninjas, 2011. http://netifera.com/research/beast/beast_DRAFT_0621.pdf.

[45] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[46] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Kurt Thomas, Vijay Eranti, Nicholas Lidzborski, Elie Bursztein, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *15th ACM Internet Measurement Conference*, 2015.

[47] Zakir Durumeric, Michael Bailey, and J. Alex Halderman. An Internet-wide view of Internet-wide scanning. In *Proc. 23rd USENIX Security Symposium*, August 2014.

[48] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proc. 14th ACM Internet Measurement Conference (IMC)*, November 2014.

[49] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proc. 22nd USENIX Security Symposium*, August 2013.

[50] Peter Eckersley and Jesse Burns. An Observatory for the SSLiverse. In *Proc. DEF CON 18*, July 2010.

[51] Young Eric. SSLeay, 1995. ftp://ftp.pl.vim.org/vol/rzm1/replay.old/libraries/SSL.eay/SSLeay-0.5.1a.tar.gz.

[52] Niels Ferguson and Bruce Schneier. A cryptographic evaluation of IPsec. *Counterpane Internet Security, Inc*, 3031, 2000.

[53] Alan Freier, Philip Karlton, and Paul Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. IETF RFC 6101, 2011.

[54] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden snfs discrete logarithm computation. In *EUROCRYPT*, 2017.

[55] M. Friedl, N. Provos, and W. Simpson. Diffie-Hellman group exchange for the Secure Shell (SSH) transport layer protocol. IETF RFC 4419, 2006.

[56] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. 10th ACM Internet Measurement Conference (IMC)*, November 2010.

[57] Sean Gallagher. Google dumps plans for OpenSSL in Chrome, takes own Boring road, 2014. http://arstechnica.com/information-technology/2014/07/google-dumps-plans-for-openssl-in-chrome-takes-own-boring-road/.

[58] Willi Geiselmann, Hubert Kopfer, Rainer Steinwandt, and Eran Tromer. Improved routing-based linear algebra for the number field sieve. In *Information Technology: Coding and Computing*, 2005.

[59] Willi Geiselmann and Rainer Steinwandt. Non-wafer-scale sieving hardware for the NFS: Another attempt to cope with 1024-bit. In *EUROCRYPT*, 2007.

[60] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, 2012.

[61] Daniel Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for TLS. IETF RFC 7919 (Draft), 2015.

[62] Daniel M Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM J. Discrete Math.*, 6(1), 1993.

[63] Daniel M Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM Journal of Discrete Math*, 1993.

[64] Robert Graham. Masscan: Designing my own crypto. Errata Security blog, December 2013. http://blog.erratasec.com/2013/12/masscan-designing-my-own-crypto.html.

[65] Robert Graham. Masscan: The entire Internet in 3 minutes. Errata Security blog, September 2013. http://blog.erratasec.com/2013/09/masscan-entire-internet-in-3-minutes.html.

[66] Matthew Green. Secure protocols in a hostile world. In *CHES 2015*, August 2015.

[67] Peter Gutmann. Cryptlib, kg_dlp.c, 2010. http://www.cypherpunks.to/~peter/cl343_beta.zip.

[68] Orman H., Purple Streak Dev., Hoffman P., and VPN Consortium. Determining strengths for public keys used for exchanging symmetric keys. IETF RFC 3766, 2004.

[69] Ryan Hamilton. QUIC discovery, 2016. https://docs.google.com/document/d/1i4m7DbrWGgXafHxwl8SwIusY2ELUe8WX258xt2LFxPM/edit#.

[70] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). IETF RFC 2409, 1998.

[71] Hashcat. http://hashcat.net.

[72] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proc. 21st USENIX Security Symposium*, August 2012.

[73] Kipp Hickman and Taher Elgamal. The SSL protocol, 1995. https://tools.ietf.org/html/draft-hickman-netscape-ssl-00.

[74] Clemens Hlauschek, Markus Gruber, Florian Fankhauser, and Christian Schanes. Prying open Pandora's box: KCI attacks against TLS. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*, 2015.

[75] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali

Kâafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium*, 2016.

[76] IANA. IPv4 address space registry. http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml.

[77] Tibor Jager, Kenneth G Paterson, and Juraj Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS 2013*, 2013.

[78] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In *17th European Symposium on Research in Computer Security*, 2012.

[79] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[80] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical invalid curve attacks on TLS-ECDH. In *European Symposium on Research in Computer Security*, 2015.

[81] Walton Jeffrey. Crypto++, 2015. https://github.com/weidai11/cryptopp/blob/48809d4e85c125814425c621d8d0d89f95405924/nbtheory.cpp#L1029.

[82] Antoine Joux and Reynald Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method. *Math. Comp.*, 72(242):953–967, 2003.

[83] Juniper TechLibrary. VPN feature guide for security devices, 2016. http://www.juniper.net/documentation/en_US/junos15.1x49/topics/reference/configuration-statement/security-edit-dh-group.html.

[84] B. Kaliski. PKCS #1 : RSA Encryption Version 1.5. IETF RFC 2313, 1998.

[85] Dan Kaminsky. Paketto simplified (1.0), November 2002. http://dankaminsky.com/2002/11/18/77/.

[86] Emilia Käsper. Fix reachable assert in SSLv2 servers. OpenSSL patch, March 2015. https://github.com/openssl/openssl/commit/86f8fb0e344d62454f8daf3e15236b2b59210756.

[87] C. Kaufman. Internet Key Exchange (IKEv2) protocol. IETF RFC 4306, 2005.

[88] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). IETF RFC 7296, 2014.

[89] Tero Kivinen and Markku Kojo. More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). IETF RFC 3526, 2003.

[90] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In *CRYPTO*, 2010.

[91] Thorsten Kleinjung, Claus Diem, Arjen K. Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In *Proc. of EUROCRYPT*, 2017.

[92] Vlastimil Klima, Ondrej Pokornỳ, and Tomáš Rosa. Attacking RSA-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[93] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. IETF RFC 7918, 2016.

[94] Robert E. Lee. Unicornscan. http://unicornscan.org.

[95] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[96] M Lepinski and S Kent. Additional Diffie-Hellman groups for use with ietf standards. IETF RFC 5114, 2008.

[97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *17th International Cryptology Conference*, 1997.

[98] Mark Lipacis. Semiconductors: Moore stress = structural industry shift. Technical report, Jefferies, 2012.

[99] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[100] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner. Internet security association and key management protocol ISAKMP. IETF RFC 2408, 1998.

[101] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In *19th ACM Conference on Computer and Communications Security*, 2012.

[102] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No need for black chambers: Testing TLS in the e-mail ecosystem at large. Technical report, 2015.

[103] C. Meadows. Analysis of the Internet key exchange protocol using the NRL protocol analyzer. In *IEEE Symposium on Security and Privacy (Oakland)*, 1999.

[104] Christopher Meyer and Jörg Schwenk. SoK: Lessons learned from SSL/TLS attacks. In *14th International Workshop on Information Security Applications*, August 2013.

[105] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Security Symposium*, 2014.

[106] Microsoft Windows Networking Team. VPN interoperability guide for Windows Server 2012 R2, 2014. https://blogs.technet.microsoft.com/networking/2014/12/26/vpn-interoperability-guide-for-windows-server-2012-r2/.

[107] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[108] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002.

[109] NIST. FIPS PUB 186-4: Digital signature standard, 2013.

[110] Andrew M. Odlyzko. The future of integer factorization. Technical report, 1995. http://www.dtc.umn.edu/~odlyzko/doc/future.of.factoring.pdf.

[111] National Cryptographic Solutions Management Office. Cryptography today, August 2015. https://web.archive.org/web/20150905185709/https://www.nsa.gov/ia/programs/suiteb_cryptography/.

[112] H. Orman. The Oakley key determination protocol. IETF RFC 2412, 1998.

[113] D Piper. The Internet IP security domain of interpretation for ISAKMP. IETF RFC 2407, 1998.

[114] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on*

Information Theory, 24(1), 1978.

[115] W Polk, R Housley, and L Bassham. Algorithms and identifiers for the internet X.509 public key infrastructure. IETF RFC 3279, 2002.

[116] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[117] M. J. Pollard. Kangaroos, Monopoly and discrete logarithms. *Journal of Cryptology*, 2000.

[118] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446, 2018.

[119] Ivan Ristić, 2016. https://www.trustworthyinternet.org/ssl-pulse/.

[120] Juliano Rizzo and Thai Duong. The CRIME attack. EKOparty Security Conference, 2012.

[121] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 Gbit/s line rate packet processing using commodity hardware: Survey and new proposals. http://luca.ntop.org/10g.pdf.

[122] Jim Roskind. QUIC design document, 2013. https://docs.google.com/a/chromium.org/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34.

[123] Antonio Sanso. OpenSSL key recovery attack on dh small subgroups, 2016. http://blog.intothesymmetry.com/2016/01/openssl-key-recovery-attack-on-dh-small.html.

[124] Oliver Schirokauer. Virtual logarithms. *J. Algorithms*, 57(2):140–147, 2005.

[125] Daniel Shanks. Class number, a theory of factorization, and genera. In *Symposia in Pure Math*, volume 20, 1969.

[126] Y Sheffer and S Fluhrer. Additional Diffie-Hellman tests for the Internet Key Exchange protocol version 2 (IKEv2). IETF RFC 6989, 2013.

[127] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm. http://cado-nfs.gforge.inria.fr/, 2017. Release 2.3.0.

[128] E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757–775, 2002.

[129] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. IETF RFC 6176, 2011.

[130] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[131] Paul C Van Oorschot and Michael J Wiener. On Diffie-Hellman key agreement with short exponents. In *EUROCRYPT*, 1996.

[132] Steven J. Vaughan-Nichols. Here comes the 100GigE Internet. *ZDNet*, 2010. http://www.zdnet.com/blog/networking/here-comes-the-100gige-internet/334.

[133] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *2nd Usenix Workshop on Electronic Commerce*, 1996.

[134] Paul Wouters. 66% of VPN's are not in fact broken, 2015. https://nohats.ca/wordpress/blog/2015/10/17/66-of-vpns-are-not-in-fact-broken/.

[135] T. Ylonen and C. Lonvick. The Secure Shell (SSH) transport layer protocol. IETF RFC 4253, 2006.

[136] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conference on Computer and Communications Security*, 2014.

[137] P. Zimmermann et al. GMP-ECM, 2012. https://gforge.inria.fr/projects/ecm.

[138] Bug 1837 - small subgroup attack, May 2016. https://bugs.exim.org/show_bug.cgi?id=1837.

[139] Exim Internet mailer, July 2015. http://www.exim.org/.

[140] Exim TLS Security, DH and standard parameters, October 2016. https://lists.exim.org/lurker/message/20161008.231103.c70b2da8.en.html.

[141] Fielded capability: End-to-end VPN SPIN 9 design review. Media leak. http://www.spiegel.de/media/media-35529.pdf.

[142] FY 2013 congressional budget justification. Media leak, 2013. http://cryptome.org/2013/08/spy-budget-fy13.pdf.

[143] Intro to the VPN exploitation process. Media leak. http://www.spiegel.de/media/media-35515.pdf.

[144] OpenSSL change log. https://www.openssl.org/news/changelog.html#x0.

[145] SPIN 15 VPN story. Media leak. http://www.spiegel.de/media/media-35522.pdf.

[146] TURMOIL VPN processing. Media leak. http://www.spiegel.de/media/media-35526.pdf.