# CMSC414 Computer and Network Security

## How Crypto Fails in Practice

Yizheng Chen | University of Maryland
surrealyz.github.io

Apr 2, 2026

Credits: original slides from Dave Levin

# Announcement

- Project 3 due on Monday, April 6
- Midterm Exam 2: Thursday, April 9
- Please bring your UMD ID card to the exam. We will check off IDs.

# Announcement

- Deadline to form Project 4 groups: Tuesday, April 7
- Each group 2 students

# RELATED PAPERS

## Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

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[‖] Paul Zimmermann[†]

[*] INRIA Paris-Rocquencourt [†] INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
[‖] Microsoft Research [‡] University of Pennsylvania [§] Johns Hopkins [¶] University of Michigan

For additional materials and contact information, visit WeakDH.org.

### ABSTRACT

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 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 are being 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.

### 1. INTRODUCTION

Diffie-Hellman key exchange is widely used to establish session keys in Internet protocols. It is the main key exchange mechanism in SSH and IPsec and a popular option in TLS. We examine how Diffie-Hellman is commonly implemented and deployed with these 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 log of the corresponding 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 [7] 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 [8], but, as far as we are aware, this is the first time they have been exploited to expose concrete vulnerabilities in real-world systems.

We were also able to compromise Diffie-Hellman for many other servers because of design and implementation flaws and configuration mistakes. These include use of composite-order subgroups in combination with short exponents, which is vulnerable to a known attack of van Oorschot and Wiener [51], and the inability of clients to properly validate Diffie-Hellman parameters without knowing the subgroup order, which TLS has no provision to communicate. We implement these attacks too and discover several vulnerable implementations.

*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


*CCS'15*, October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
DOI: http://dx.doi.org/10.1145/2810103.2813707.

## The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

Martin Georgiev
The University of Texas at Austin

Subodh Iyengar
Stanford University

Suman Jana
The University of Texas at Austin

Rishita Anubhai
Stanford University

Dan Boneh
Stanford University

Vitaly Shmatikov
The University of Texas at Austin

### ABSTRACT

SSL (Secure Sockets Layer) is the de facto standard for secure Internet communications. Security of SSL connections against an active network attacker depends on correctly validating public-key certificates presented when the connection is established.

We demonstrate that SSL certificate validation is completely broken in many security-critical applications and libraries. Vulnerable software includes Amazon's EC2 Java library and all cloud clients based on it; Amazon's and PayPal's merchant SDKs responsible for transmitting payment details from e-commerce sites to payment gateways; integrated shopping carts such as osCommerce, ZenCart, Ubercart, and PrestaShop; AdMob code used by mobile websites; Chase mobile banking and several other Android apps and libraries; Java Web-services middleware—including Apache Axis, Axis 2, Codehaus XFire, and Pusher library for Android—and *all* applications employing this middleware. Any SSL connection from any of these programs is insecure against a man-in-the-middle attack.

The root causes of these vulnerabilities are badly designed APIs of SSL implementations (such as JSSE, OpenSSL, and GnuTLS) and data-transport libraries (such as cURL) which present developers with a confusing array of settings and options. We analyze perils and pitfalls of SSL certificate validation in software based on these APIs and present our recommendations.

### Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

### Keywords

SSL, TLS, HTTPS, public-key infrastructure, public-key certificates, security vulnerabilities

### 1. INTRODUCTION

Originally deployed in Web browsers, SSL (Secure Sockets Layer) has become the de facto standard for secure Internet communications. The main purpose of SSL is to provide end-to-end security against an active, man-in-the-middle attacker. Even if the network is completely compromised—DNS is poisoned, access points and routers are controlled by the adversary, etc.—SSL is intended to guarantee confidentiality, authenticity, and integrity for communications between the client and the server.

Authenticating the server is a critical part of SSL connection establishment.[1] This authentication takes place during the SSL handshake, when the server presents its public-key certificate. In order for the SSL connection to be secure, the client must carefully verify that the certificate has been issued by a valid certificate authority, has not expired (or been revoked), the name(s) listed in the certificate match(es) the name of the domain that the client is connecting to, and perform several other checks [14, 15].

SSL implementations in Web browsers are constantly evolving through "penetrate-and-patch" testing, and many SSL-related vulnerabilities in browsers have been repaired over the years. SSL, however, is also widely used in *non-browser software* whenever secure Internet connections are needed. For example, SSL is used for (1) remotely administering cloud-based virtual infrastructure and sending local data to cloud-based storage, (2) transmitting customers' payment details from e-commerce servers to payment processors such as PayPal and Amazon, (3) logging instant messenger clients into online services, and (4) authenticating servers to mobile applications on Android and iOS.

These programs usually do not implement SSL themselves. Instead, they rely on SSL libraries such as OpenSSL, GnuTLS, JSSE, CryptoAPI, etc., as well as higher-level data-transport libraries, such as cURL, Apache HttpClient, and *urllib*, that act as wrappers around SSL libraries. In software based on Web services, there is an additional layer of abstraction introduced by Web-services middleware such as Apache Axis, Axis 2, or Codehaus XFire.

**Our contributions.** We present an in-depth study of SSL connection authentication in non-browser software, focusing on how diverse applications and libraries on Linux, Windows, Android, and iOS validate SSL server certificates. We use both white- and black-box techniques to discover vulnerabilities in validation logic. Our main conclusion is that *SSL certificate validation is completely broken in many critical software applications and libraries*. When presented with self-signed and third-party certificates—including a certificate issued by a legitimate authority to a domain called `AllYourSSLAreBelongTo.us` —they establish SSL connections and send their secrets to a man-in-the-middle attacker.

[1] SSL also supports client authentication, but we do not analyze it in this paper.


*CCS'12*, October 16–18, 2012, Raleigh, North Carolina, USA.

# POOR PROGRAMING

## An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

**ABSTRACT**

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. In this paper, we ask whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We develop program analysis techniques to automatically check programs on the Google Play marketplace, and find that 10,327 out of 11,748 applications that use cryptographic APIs – 88% overall – make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediations based on our analysis towards improving overall cryptographic security in Android applications.

**Categories and Subject Descriptors**

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

**General Terms**

Android program slicing, Misuse of cryptographic primitives

**Keywords**

Software Security, Program Analysis

### 1 Introduction

Developers use cryptographic primitives like block ciphers and message authenticate codes (MACs) to secure data and communications. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way invariably leads to trouble.

In this paper, we ask whether developers know how to use cryptographic APIs in a cryptographically correct fashion. In particular, given code that type-checks and compiles, does the implemented code use cryptographic primitives correctly to achieve typical definitions of security? We assume that developers who use cryptography in their applications make this choice consciously. After all, a developer would not likely try to encrypt or authenticate data that they did not believe needed securing.


CCS'13, November 04 - 08 2013, Berlin, Germany

http://dx.doi.org/10.1145/2508859.2516693.

We focus on two well-known security standards: security against chosen plaintext attacks (IND-CPA) and cracking resistance. For each definition of security, there is a generally accepted right and wrong way to do things. For example, electronic code book (ECB) mode should only be used by cryptographic experts. This is because identical plaintext blocks encrypt to identical ciphertext blocks, thus rendering ECB non-IND-CPA secure. When creating a password hash, a unique salt should be chosen to make password cracking more computationally expensive.

We focus on the Android platform, which is attractive for three reasons. First, Android applications run on smart phones, and smart phones manage a tremendous amount of personal information such as passwords, location, and social network data. Second, Android is closely related to Java, and Java's cryptographic API is stable. For example, the `Cipher` API which provides access to various encryption schemes has been unmodified since Java 1.4 was released in 2002. Third, the large number of available Android applications allows us to perform our analysis on a large dataset, thus gaining insight into how application developers use cryptographic primitives.

One approach for checking cryptographic implementations would be to adapt verification-based tools like the Microsoft Crypto Verification Kit [7], Mur$\varphi$ [22], and others. The main advantage of verification-based approaches is that they provide strong guarantees. However, they are also heavy-weight, require significant expertise, and require manual effort. The sum of these three limitations make the tools inappropriate for large-scale experiments, or for use by day-to-day developers who are not cryptographers.

Instead, we adopt a light-weight static analysis approach that checks for common flaws. Our tool, called CryptoLint, is based upon the Androguard Android program analysis framework [12]. The main new idea in CryptoLint is to use static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and the cryptographic operations themselves. CryptoLint takes a raw Android binary, disassembles it, and checks for typical cryptographic misuses quickly and accurately. These characteristics make CryptoLint appropriate for use by developers, app store operators, and security-conscious users.

Using CryptoLint, we performed a study on crypto-

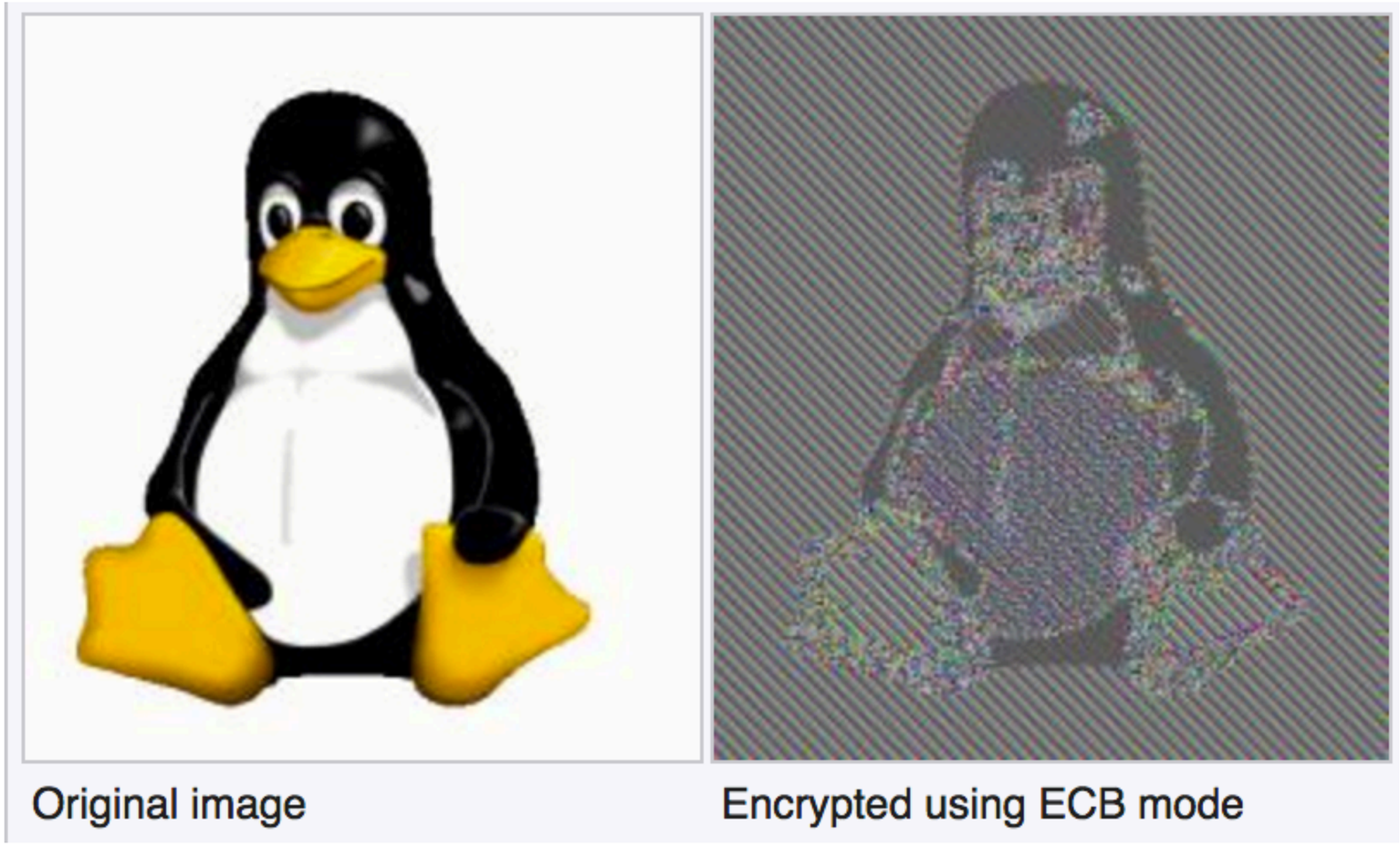
**Rule 1:** Do not use ECB mode for encryption. [6]
**Rule 2:** Do not use a non-random IV for CBC encryption. [6, 23]
**Rule 3:** Do not use constant encryption keys.
**Rule 4:** Do not use constant salts for PBE. [2, 5]
**Rule 5:** Do not use fewer than 1,000 iterations for PBE. [2, 5]
**Rule 6:** Do not use static seeds to seed `SecureRandom(·)`.

*CryptoLint tool to perform static analysis on Android apps to detect how they are using crypto libraries*

# CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

| | # apps | violated rule |
|---|---|---|
| 48% | 5,656 | Uses ECB (BouncyCastle default) (R1) |
| 31% | 3,644 | Uses constant symmetric key (R3) |
| 17% | 2,000 | Uses ECB (Explicit use) (R1) |
| 16% | 1,932 | Uses constant IV (R2) |
| | 1,636 | Used iteration count < 1,000 for PBE(R5) |
| 14% | 1,629 | Seeds SecureRandom with static (R6) |
| | 1,574 | Uses static salt for PBE (R4) |
| 12% | 1,421 | No violation |



Original image

Encrypted using ECB mode

NEVER use ECB
(but over 50% of Android apps do)

# BOUNCYCASTLE DEFAULTS

- BouncyCastle is a library that conforms to Java's `Cipher` interface:

```
Cipher c =
   Cipher.getInstance("AES/CBC/PKCS5Padding");

// Ultimately end up wrapping a ByteArrayOutputStream
// in a CipherOutputStream
```

- Java documentation specifies:

> If no mode or padding is specified, provider-specific default values for the mode and padding scheme are used. For example, the `SunJCE` provider uses `ECB` as the default mode, and `PKCS5Padding` as the default padding scheme for `DES`, `DES-EDE` and `Blowfish` ciphers

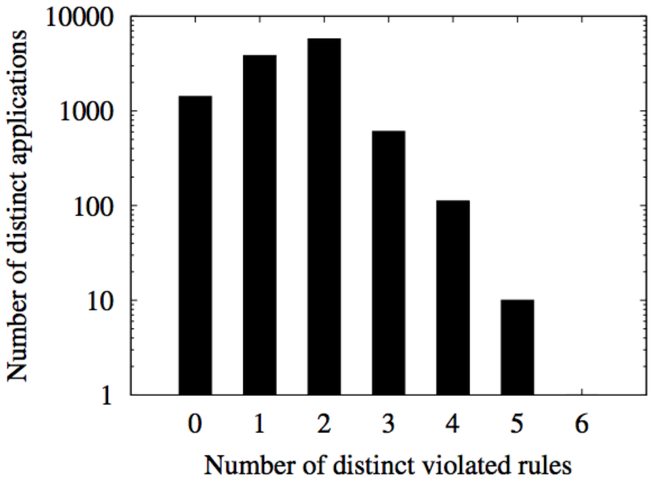
| #Occurences | Symmetric encryption scheme |
|---|---|
| 5878 | AES/CBC/PKCS5Padding |
| 4803 | AES * |
| 1151 | DES/ECB/NoPadding |
| 741 | DES * |
| 501 | DESede * |
| 473 | DESede/ECB/PKCS5Padding |
| 468 | AES/CBC/NoPadding |
| 443 | AES/ECB/PKCS5Padding |
| 235 | AES/CBC/PKCS7Padding |
| 221 | DES/ECB/PKCS5Padding |
| 220 | AES/ECB/NoPadding |
| 205 | DES/CBC/PKCS5Padding |
| 155 | AES/ECB/PKCS7Padding |
| 104 | AES/CFB8/NoPadding |

**Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.**

# CRYPTO MISUSE IN ANDROID APPS

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

| | # apps | violated rule |
|---|---|---|
| 48% | 5,656 | Uses ECB (BouncyCastle default) (R1) |
| 31% | 3,644 | Uses constant symmetric key (R3) |
| 17% | 2,000 | Uses ECB (Explicit use) (R1) |
| 16% | 1,932 | Uses constant IV (R2) |
| | 1,636 | Used iteration count < 1,000 for PBE(R5) |
| 14% | 1,629 | Seeds SecureRandom with static (R6) |
| | 1,574 | Uses static salt for PBE (R4) |
| 12% | 1,421 | No violation |




A failure of the programmers to **know the tools** they use

A failure of library writers to **provide safe defaults**

# MISUSING CRYPTO

Avoid shooting yourself in the foot:

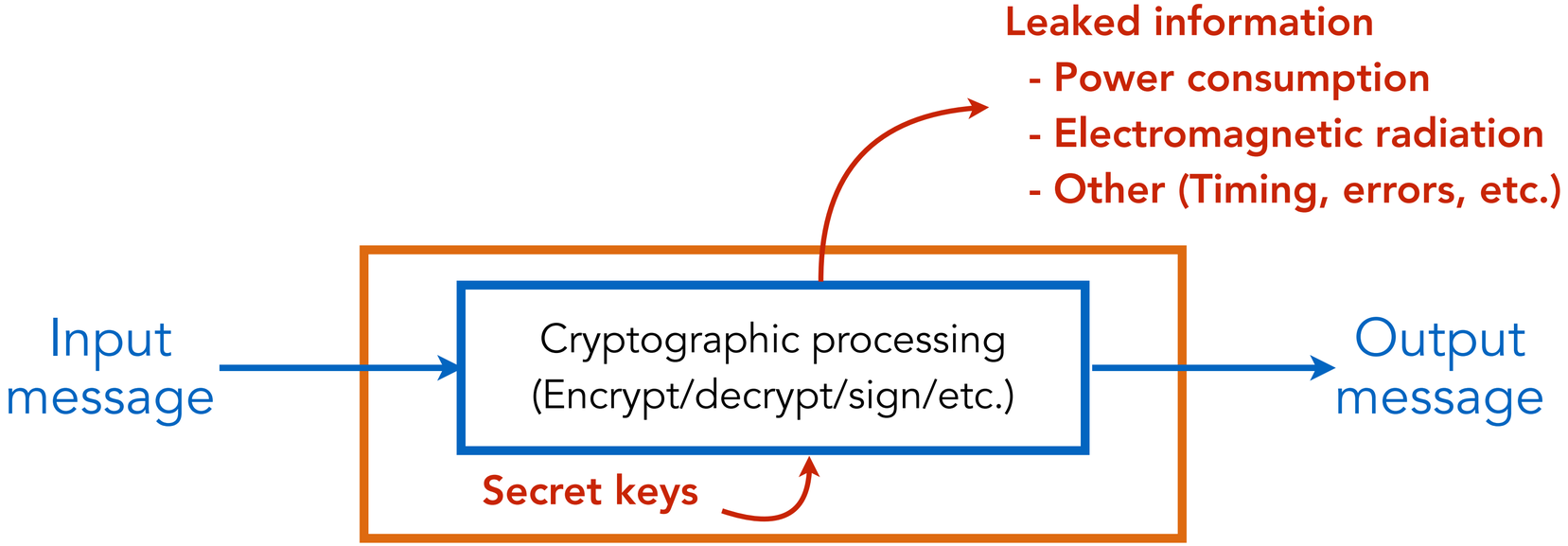
- Do not **roll your own** cryptographic mechanisms
  - Takes peer review
  - Apply Kerkhoff's principle
- Do not ***misuse*** existing crypto
- Do not even ***implement*** the underlying crypto

# WHY NOT IMPLEMENT AES/RSA YOURSELF?

- Not talking about creating a brand new crypto scheme, just implementing one that's already widely accepted and used.
- Kerkhoff's principle: these are all open standards; should be implementable.
- Potentially buggy/incorrect code, but so might be others' implementations (viz. OpenSSL bugs, poor defaults in Bouncy castles, etc.)
- So why not implement it yourself?

# SIDE-CHANNEL ATTACKS

- Cryptography concerns the *theoretical* difficulty in breaking a cipher




- But what about the information that a particular *implementation* could leak?
  - Attacks based on these are "**side-channel attacks**"

# SIMPLE POWER ANALYSIS (SPA)

- Interpret *power traces* taken during a cryptographic operation
- Simple power analysis can reveal the sequence of instructions executed

# SPA ON DES

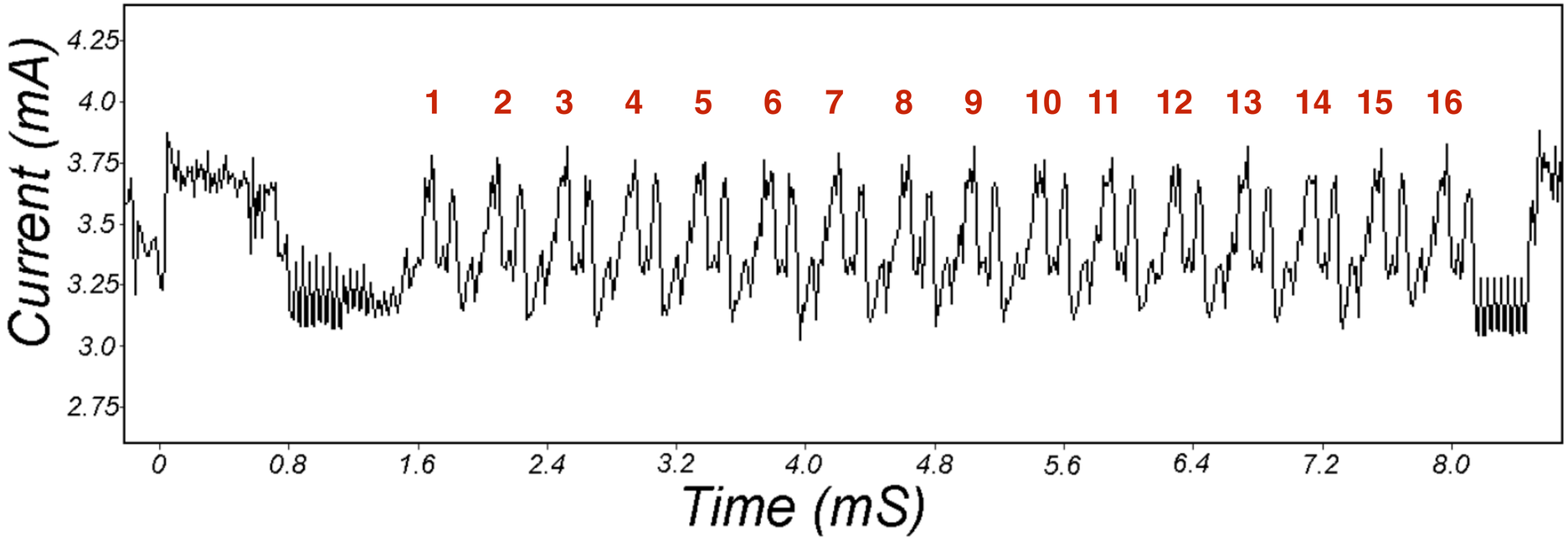



**Figure 1:** SPA trace showing an entire DES operation.

Overall operation clearly visible:

Can identify the **16 rounds of DES**

# SPA ON DES

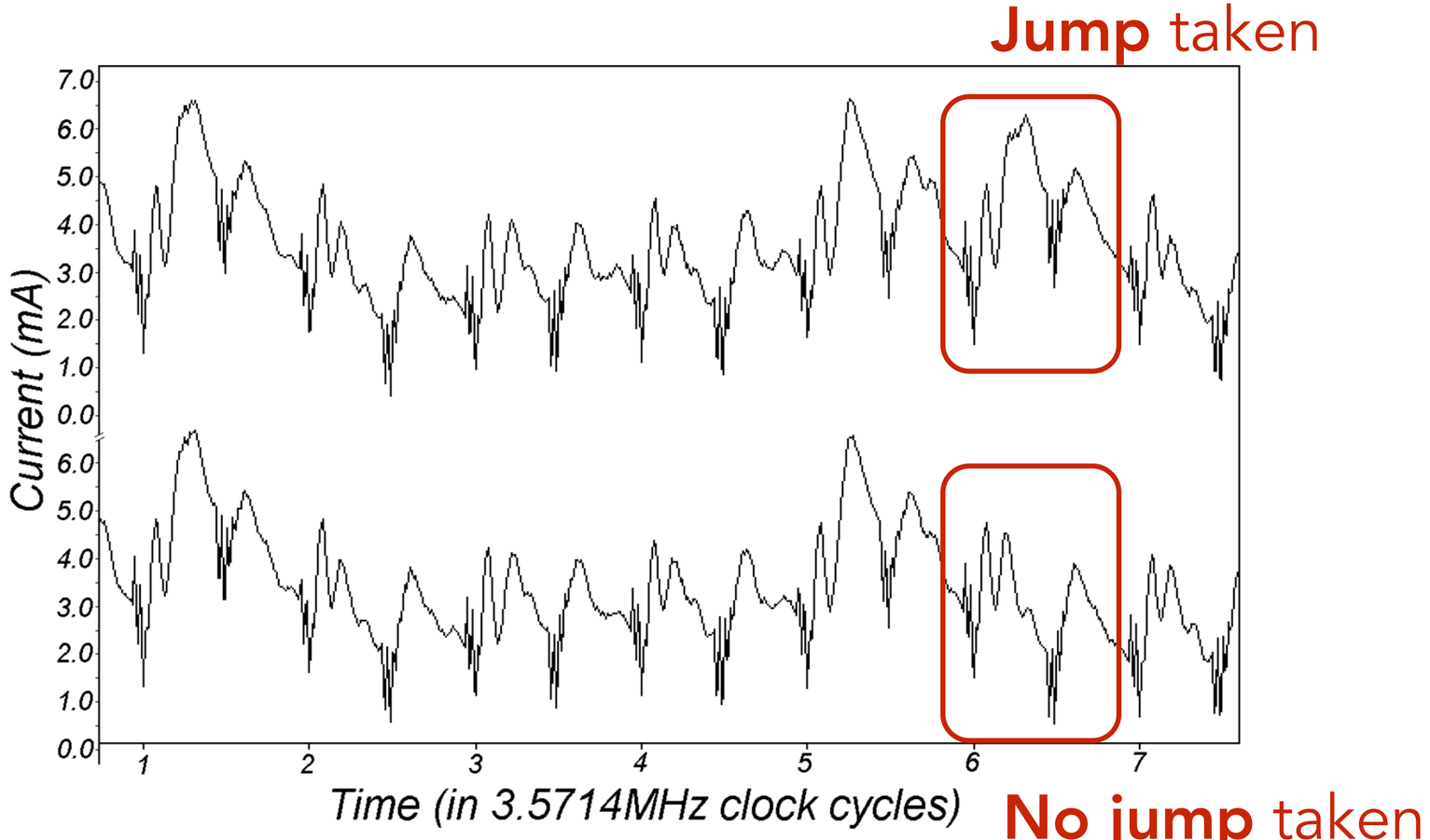



**Figure 3:** SPA trace showing individual clock cycles.

Specific **instructions** are also discernible

# HIGH-LEVEL IDEA

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

What if branch 0
- took longer? (timing attacks)
- gave off more heat?
- made more noise?
- …

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

# DIFFERENTIAL POWER ANALYSIS (DPA)

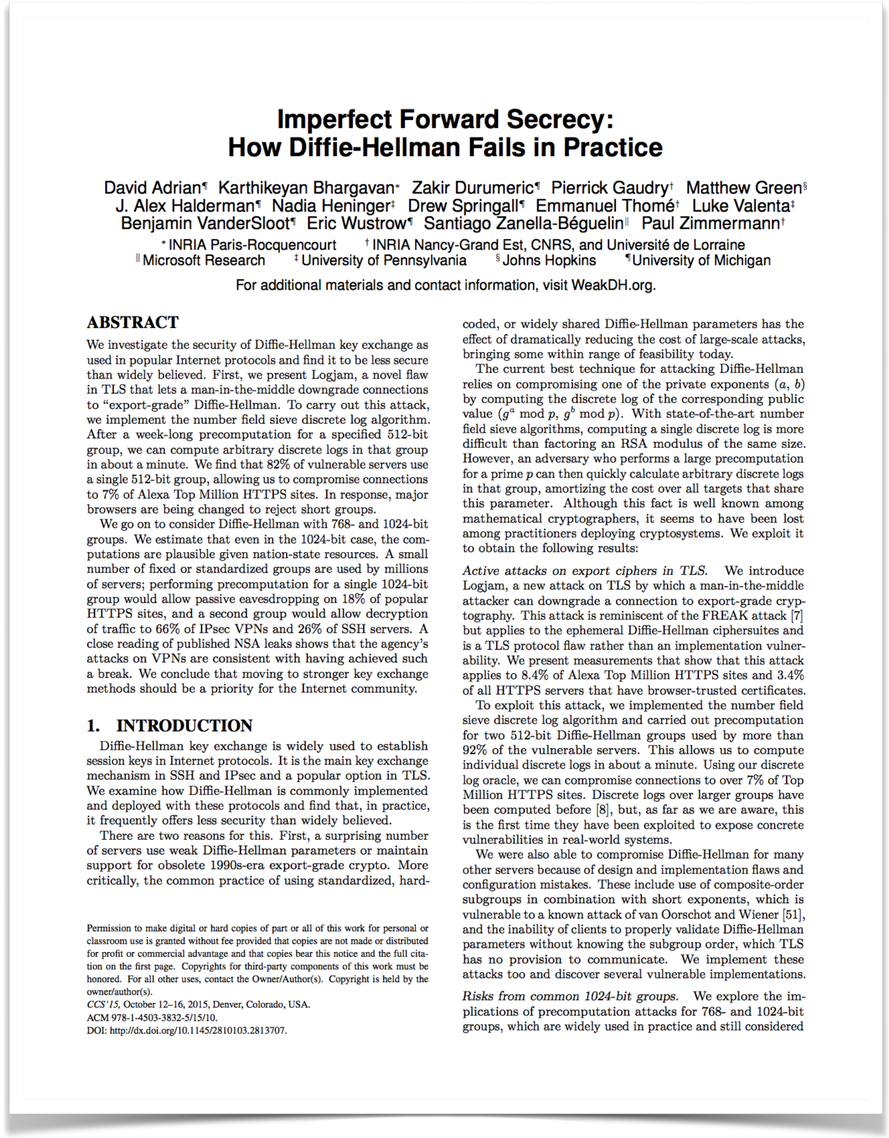
- SPA just visually inspects a single run
- DPA runs iteratively and reactively
  - Get multiple samples
  - Based on these, construct new plaintext messages as inputs, and repeat

# MITIGATING SUCH ATTACKS

- Hide information by making the execution paths depend on the inputs as little as possible
  - Have to *give up some optimizations* that depend on particular bit values in keys
    - Some Chinese Remainder Theorem (CRT) optimizations permitted remote timing attacks on SSL servers

- The crypto community should seek to design cryptosystems under the assumption that some information is going to leak

# POOR POLICIES



**Imperfect Forward Secrecy:**
**How Diffie-Hellman Fails in Practice**

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[‖] Paul Zimmermann[†]

[*]INRIA Paris-Rocquencourt [†]INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
[‖]Microsoft Research [‡]University of Pennsylvania [§]Johns Hopkins [¶]University of Michigan

For additional materials and contact information, visit WeakDH.org.

**ABSTRACT**

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 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 are being 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.

**1. INTRODUCTION**

Diffie-Hellman key exchange is widely used to establish session keys in Internet protocols. It is the main key exchange mechanism in SSH and IPsec and a popular option in TLS. We examine how Diffie-Hellman is commonly implemented and deployed with these 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 log of the corresponding 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 [7] 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 [8], but, as far as we are aware, this is the first time they have been exploited to expose concrete vulnerabilities in real-world systems.

We were also able to compromise Diffie-Hellman for many other servers because of design and implementation flaws and configuration mistakes. These include use of composite-order subgroups in combination with short exponents, which is vulnerable to a known attack of van Oorschot and Wiener [51], and the inability of clients to properly validate Diffie-Hellman parameters without knowing the subgroup order, which TLS has no provision to communicate. We implement these attacks too and discover several vulnerable implementations.

*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


*CCS'15,* October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
DOI: http://dx.doi.org/10.1145/2810103.2813707.

| Source | Popularity | Prime |
|---|---|---|
| Apache | 82% | `9fdb8b8a004544f0045f1737d0ba2e0b`<br>`274cdf1a9f588218fb435316a16e3741`<br>`71fd19d8d8f37c39bf863fd60e3e3006`<br>`80a3030c6e4c3757d08f70e6aa871033` |
| mod_ssl | 10% | `d4bcd52406f69b35994b88de5db89682`<br>`c8157f62d8f33633ee5772f11f05ab22`<br>`d6b5145b9f241e5acc31ff090a4bc711`<br>`48976f76795094e71e7903529f5a824b` |
| (*others*) | 8% | (463 distinct primes) |

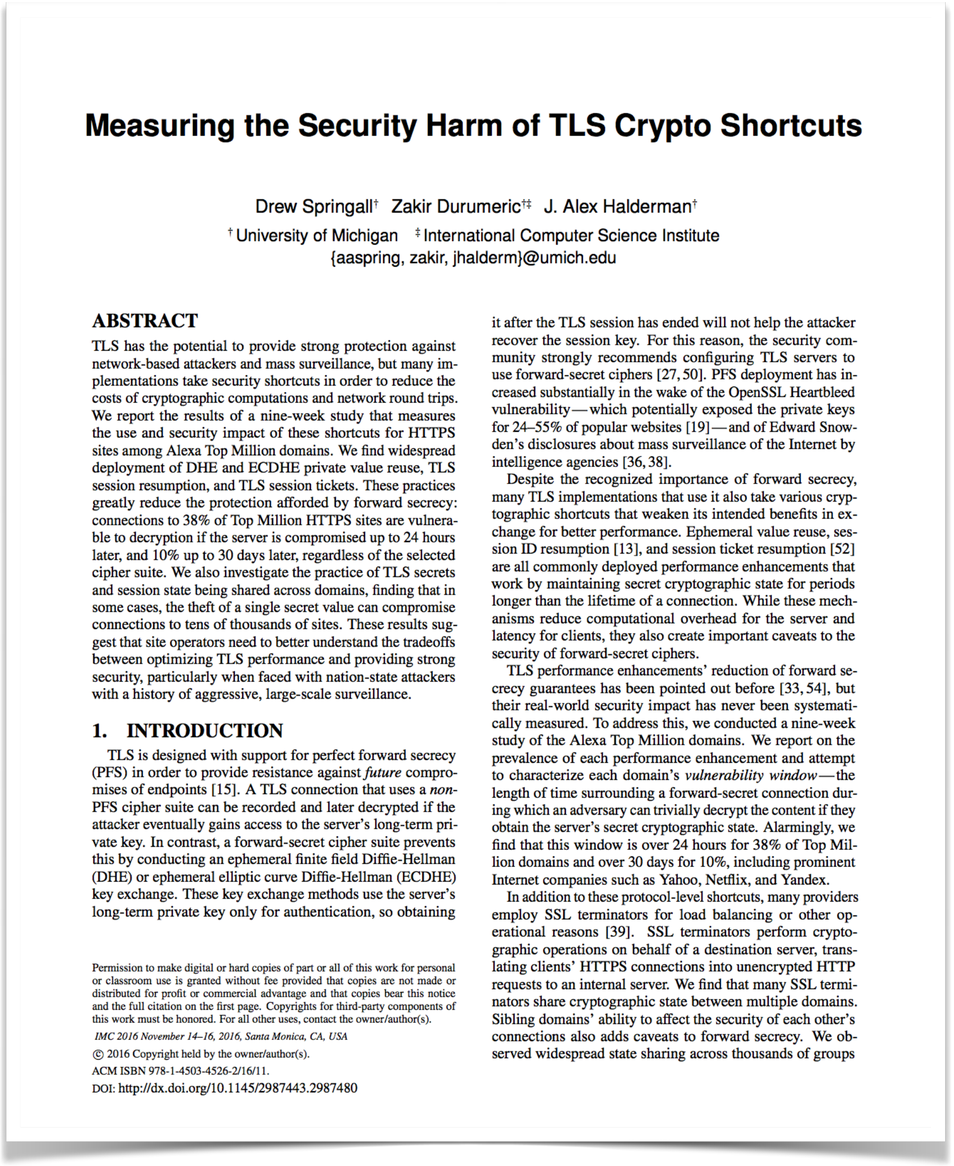
Table 1: **Top 512-bit DH primes for TLS.** 8.4% of Alexa Top 1M HTTPS domains allow DHE_EXPORT, of which 92.3% use one of the two most popular primes, shown here.

*"After a week-long **precomputation** 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."*

# FORWARD SECRECY

- Compromising a long-term key should not compromise past session keys

# UNSAFE OPTIMIZATIONS



**Measuring the Security Harm of TLS Crypto Shortcuts**

Drew Springall† Zakir Durumeric†‡ J. Alex Halderman†
†University of Michigan ‡International Computer Science Institute
{aaspring, zakir, jhalderm}@umich.edu

**ABSTRACT**

TLS has the potential to provide strong protection against network-based attackers and mass surveillance, but many implementations take security shortcuts in order to reduce the costs of cryptographic computations and network round trips. We report the results of a nine-week study that measures the use and security impact of these shortcuts for HTTPS sites among Alexa Top Million domains. We find widespread deployment of DHE and ECDHE private value reuse, TLS session resumption, and TLS session tickets. These practices greatly reduce the protection afforded by forward secrecy: connections to 38% of Top Million HTTPS sites are vulnerable to decryption if the server is compromised up to 24 hours later, and 10% up to 30 days later, regardless of the selected cipher suite. We also investigate the practice of TLS secrets and session state being shared across domains, finding that in some cases, the theft of a single secret value can compromise connections to tens of thousands of sites. These results suggest that site operators need to better understand the tradeoffs between optimizing TLS performance and providing strong security, particularly when faced with nation-state attackers with a history of aggressive, large-scale surveillance.

**1. INTRODUCTION**

TLS is designed with support for perfect forward secrecy (PFS) in order to provide resistance against *future* compromises of endpoints [15]. A TLS connection that uses a *non*-PFS cipher suite can be recorded and later decrypted if the attacker eventually gains access to the server's long-term private key. In contrast, a forward-secret cipher suite prevents this by conducting an ephemeral finite field Diffie-Hellman (DHE) or ephemeral elliptic curve Diffie-Hellman (ECDHE) key exchange. These key exchange methods use the server's long-term private key only for authentication, so obtaining it after the TLS session has ended will not help the attacker recover the session key. For this reason, the security community strongly recommends configuring TLS servers to use forward-secret ciphers [27, 50]. PFS deployment has increased substantially in the wake of the OpenSSL Heartbleed vulnerability — which potentially exposed the private keys for 24–55% of popular websites [19] — and of Edward Snowden's disclosures about mass surveillance of the Internet by intelligence agencies [36, 38].

Despite the recognized importance of forward secrecy, many TLS implementations that use it also take various cryptographic shortcuts that weaken its intended benefits in exchange for better performance. Ephemeral value reuse, session ID resumption [13], and session ticket resumption [52] are all commonly deployed performance enhancements that work by maintaining secret cryptographic state for periods longer than the lifetime of a connection. While these mechanisms reduce computational overhead for the server and latency for clients, they also create important caveats to the security of forward-secret ciphers.

TLS performance enhancements' reduction of forward secrecy guarantees has been pointed out before [33, 54], but their real-world security impact has never been systematically measured. To address this, we conducted a nine-week study of the Alexa Top Million domains. We report on the prevalence of each performance enhancement and attempt to characterize each domain's *vulnerability window* — the length of time surrounding a forward-secret connection during which an adversary can trivially decrypt the content if they obtain the server's secret cryptographic state. Alarmingly, we find that this window is over 24 hours for 38% of Top Million domains and over 30 days for 10%, including prominent Internet companies such as Yahoo, Netflix, and Yandex.

In addition to these protocol-level shortcuts, many providers employ SSL terminators for load balancing or other operational reasons [39]. SSL terminators perform cryptographic operations on behalf of a destination server, translating clients' HTTPS connections into unencrypted HTTP requests to an internal server. We find that many SSL terminators share cryptographic state between multiple domains. Sibling domains' ability to affect the security of each other's connections also adds caveats to forward secrecy. We observed widespread state sharing across thousands of groups


*IMC 2016 November 14–16, 2016, Santa Monica, CA, USA*

ACM ISBN 978-1-4503-4526-2/16/11.
DOI: http://dx.doi.org/10.1145/2987443.2987480

## *TLS session ticket resumption*

*Session ticket: session keys and other data to resume the session*

*Server sends an "opaque" ticket (encrypted with the Session Ticket Encryption Key, STEK)*

*Client sends the encrypted session ticket during handshake; server uses the STEK to recover it and pick up in one round-trip of communication*

# UNSAFE OPTIMIZATIONS

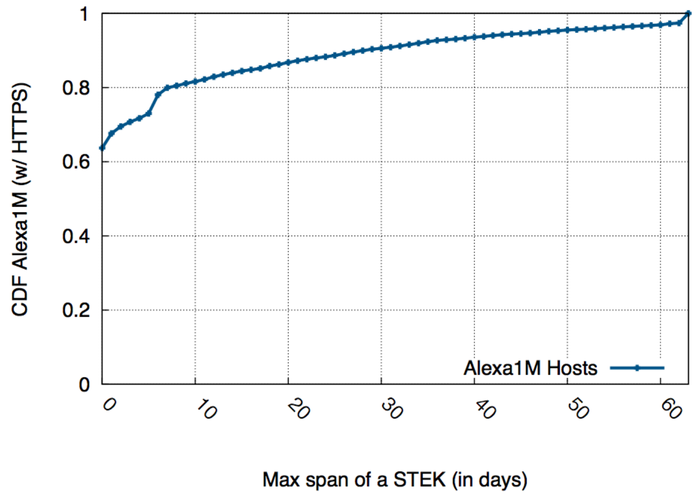



Figure 3: **STEK Lifetime**—TLS connections cannot achieve forward secrecy until the STEK (the key used by the server to encrypt the session ticket) is discarded.

*Incentive to hold onto STEKs (lower RTTs)*

*But they're holding onto them long enough for nation-states to recover them*