The legacy of export-grade cryptography in the 21st century

Nadia Heninger and J. Alex Halderman

University of Pennsylvania University of Michigan

June 9, 2016

International Traffic in Arms Regulations

April 1, 1992 version

Category XIII--Auxiliary Military Equipment ...

(b) Information Security Systems and equipment, cryptographic devices, software, and components specifically designed or modified therefore, including:

(1) Cryptographic (including key management) systems, equipment, assemblies, modules, integrated circuits, components or software with the capability of maintaining secrecy or confidentiality of information or information systems, except cryptographic equipment and software as follows:

(i) Restricted to decryption functions specifically designed to allow the execution of copy protected software, provided the decryption functions are not user-accessible.

(ii) Specially designed, developed or modified for use in machines for banking or money transactions, and restricted to use only in such transactions. Machines for banking or money transactions include automatic teller machines, self-service statement printers, point of sale terminals or equipment for the encryption of interbanking transactions.

...

Timeline of US cryptography export control

- Pre-1994: Encryption software requires individual export license as a munition.
- 1994: US State Department amends ITAR regulations to allow export of approved software to approved countries without individual licenses. 40-bit symmetric cryptography was understood to be approved under this scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Bernstein v. United States; California judge rules ITAR regulations are unconstitutional because "code is speech"
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.

Commerce Control List: Category 5 - Info. Security

(May 21, 2015 version)

a.1.a. A symmetric algorithm employing a key length in excess of 56-bits; or

a.1.b. An asymmetric algorithm where the security of the algorithm is based on any of the following:

a.1.b.1. Factorization of integers in excess of 512 bits (e.g., RSA);

a.1.b.2. Computation of discrete logarithms in a multiplicative group of a finite field of size greater than 512 bits (e.g., Diffie-Hellman over Z/pZ); or

a.1.b.3. Discrete logarithms in a group other than mentioned in 5A002.a.1.b.2 in excess of 112 bits (e.g., Diffie-Hellman over an elliptic curve);

a.2. Designed or modified to perform cryptanalytic functions;

"The government must be wary of suffocating [the encryption software] industry with regulation in the new digital age, but we must be able to strike a balance between the legitimate concerns of the law enforcement community and the needs of the marketplace." — U.S. Vice President Al Gore, September 1997

"The government must be wary of suffocating [the encryption software] industry with regulation in the new digital age, but we must be able to strike a balance between the legitimate concerns of the law enforcement community and the needs of the marketplace." — U.S. Vice President Al Gore, September 1997

"Because, if, in fact, you can't crack that [encryption] at all, government can't get in, then everybody is walking around with a Swiss bank account in their pocket – right? So there has to be some concession to the need to be able to get into that information somehow." — President Obama, March 2016

Historical experiment: How did this "compromise" work out for us?

Updated timeline of export control

- 1994: ITAR regulatory scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.
- ...

Updated timeline of export control

- 1994: ITAR regulatory scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.
- ...
- March 2015: **FREAK** attack; 10% of popular sites vulnerable.
- May 2015: **Logjam** attack; 8% of popular sites vulnerable.
- March 2016: **DROWN** attack; 25% of popular sites vulnerable.

The FREAK attack

A Messy State of the Union: Taming the Composite State Machines of TLS Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Jean Karim Zinzindohoue *Oakland 2015*

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

Public Key

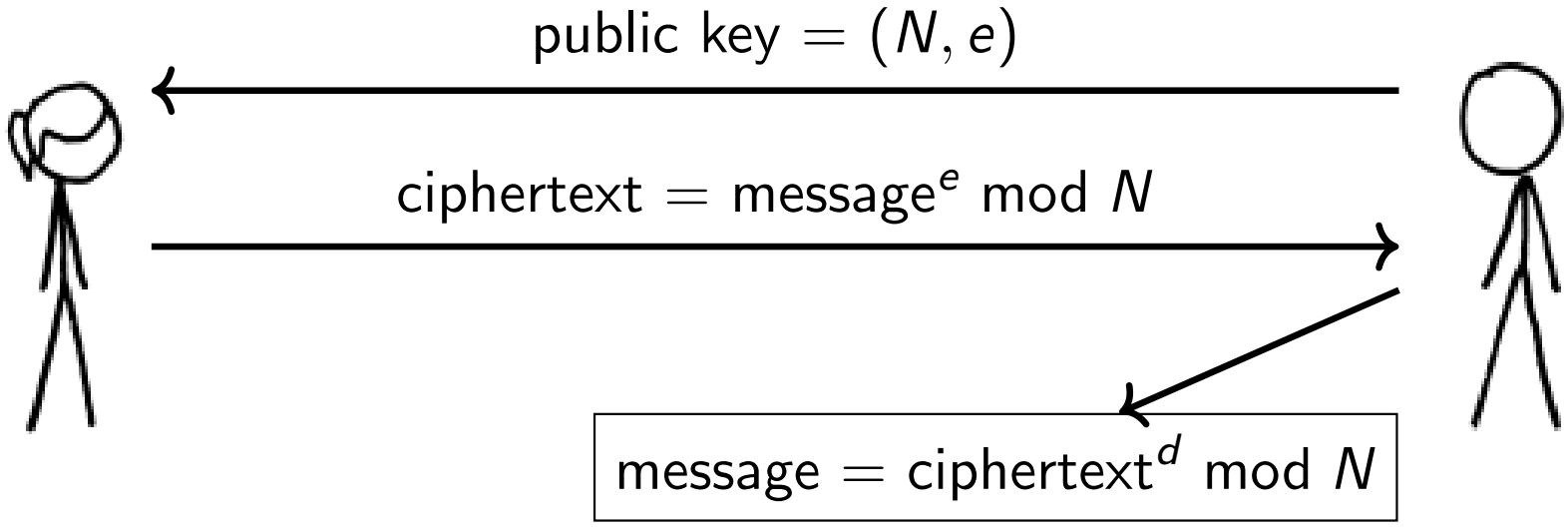
$N = pq$ modulus

e encryption exponent

Private Key

p, q primes

d decryption exponent
$(d = e^{-1} \bmod (p-1)(q-1))$



RSA cryptanalysis: computational problems

Factoring

Problem: Given N, compute its prime factors.

- Computationally equivalent to computing private key d.
- Factoring is in NP and coNP $\rightarrow$ not NP-complete (unless P=NP or similar).

eth roots mod N

Problem: Given N, e, and c, compute x such that $x^e \equiv c \bmod N$.

- Equivalent to decrypting an RSA-encrypted ciphertext.
- Not known whether it is equivalent to factoring.

Bank of America | Home | Pe

Bank of America Corporation [US] https://www.bankofamerica.com

Personal | Small Business | Wealth Management | Businesses & Institutions

of America

Locations | Contact Us | Help | En español | Search Bank of America

Online ID | Sign In

is Online ID | Enroll

ount location

Bank | Borrow | Invest | Protect | Pl

$100

bonus cash back offer

Offer Details

BankAmericard Cash Rewards™ credit card

1% cash back **everywhere, every time**

2% cash back on **groceries**

3% cash back on **gas**

Grocery/gas bonus rewards on $1,500 in combined purchases each quarter.

n for: Select a state | Go

Website Ad

Banking

Secure access to your money anytime, anywhere.

Online Bill Pay

The fast convenient way to pay your bills.

Donating homes to veterans

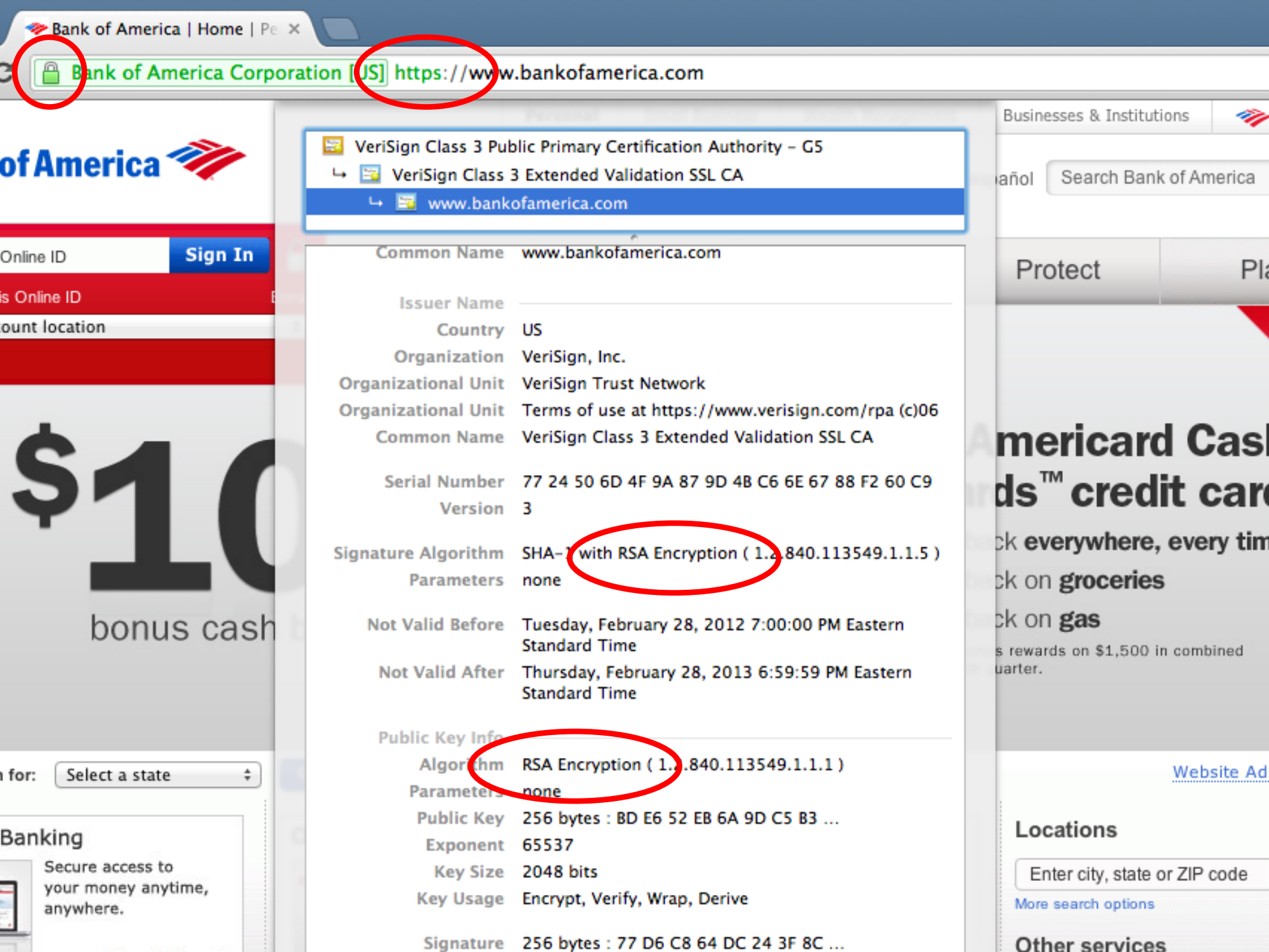
We've committed to donate 1000 properties to veterans and first responders.

Locations

Enter city, state or ZIP code

More search options

Other services


Bank of America | Home
Bank of America Corporation [US] https://www.bankofamerica.com
VeriSign Class 3 Public Primary Certification Authority - G5
VeriSign Class 3 Extended Validation SSL CA
www.bankofamerica.com
Common Name www.bankofamerica.com
Issuer Name
Country US
Organization VeriSign, Inc.
Organizational Unit VeriSign Trust Network
Organizational Unit Terms of use at https://www.verisign.com/rpa (c)06
Common Name VeriSign Class 3 Extended Validation SSL CA
Serial Number 77 24 50 6D 4F 9A 87 9D 4B C6 6E 67 88 F2 60 C9
Version 3
Signature Algorithm SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)
Parameters none
Not Valid Before Tuesday, February 28, 2012 7:00:00 PM Eastern Standard Time
Not Valid After Thursday, February 28, 2013 6:59:59 PM Eastern Standard Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters none
Public Key 256 bytes : BD E6 52 EB 6A 9D C5 B3 ...
Exponent 65537
Key Size 2048 bits
Key Usage Encrypt, Verify, Wrap, Derive
Signature 256 bytes : 77 D6 C8 64 DC 24 3F 8C ...
Businesses & Institutions
Search Bank of America
Sign In
Protect
bonus cash
Banking
Secure access to your money anytime, anywhere.
Locations
Enter city, state or ZIP code
More search options
Other services

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

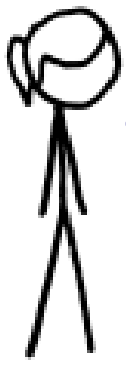
certificate = **RSA pubkey** k_{2048} + CA signatures

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]



certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{mc}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{m_c}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\text{Auth}_{k_{m_s}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\mathrm{Auth}_{k_{m_s}}(\mathrm{dialog})$

$\mathrm{Enc}_{k_e}(\mathrm{request})$

Question: How do you selectively weaken a protocol based on RSA?

Question: How do you selectively weaken a protocol based on RSA?

Export answer: Optionally use a small RSA key.

TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

TLS RSA Export Key Exchange

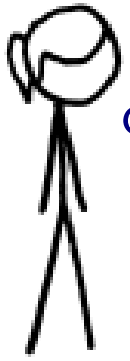
client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**



TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

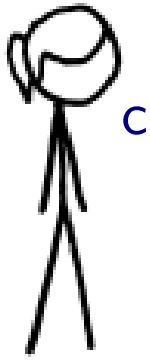
server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

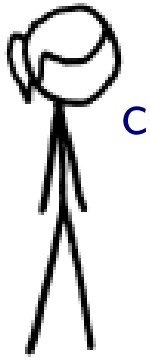
client finished: Auth$_{k_{mc}}$(dialog)



KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Export Key Exchange



client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

client finished: Auth$_{k_{mc}}$(dialog)

server finished: Auth$_{k_{ms}}$(dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

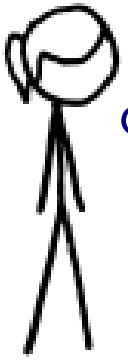
KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]



certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: Auth$_{k_{m_c}}$(dialog)

server finished: Auth$_{k_{m_s}}$(dialog)

Enc$_{k_e}$(request)

RSA export cipher suites in TLS

In March 2015, export cipher suites supported by 36.7% of the 14 million sites serving browser-trusted certificates!

`TLS_RSA_EXPORT_WITH_RC4_40_MD5`
`TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5`
`TLS_RSA_EXPORT_WITH_DES40_CBC_SHA`

Totally insecure, but no modern client would negotiate export ciphers. ... right?

Tracking the Freak Attack Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman `freakattack.com`

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

[. . . RSA . . .]

FREAK: MITM downgrade attack to export RSA

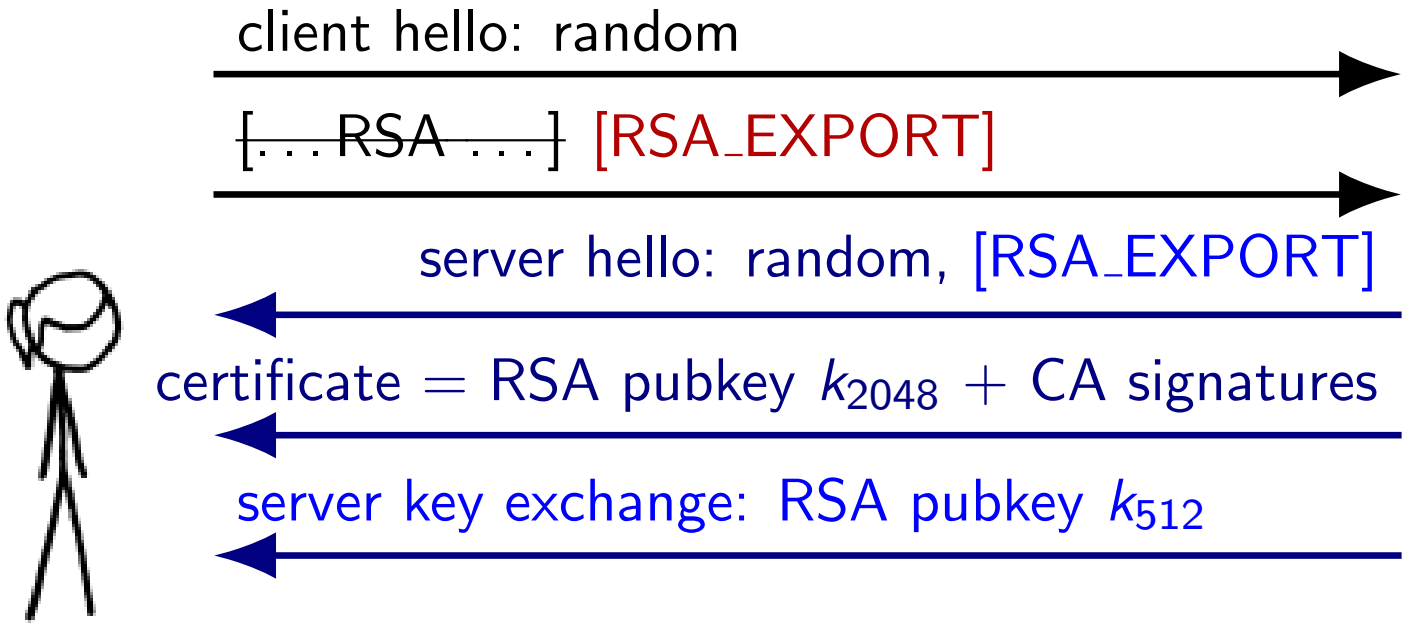
Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

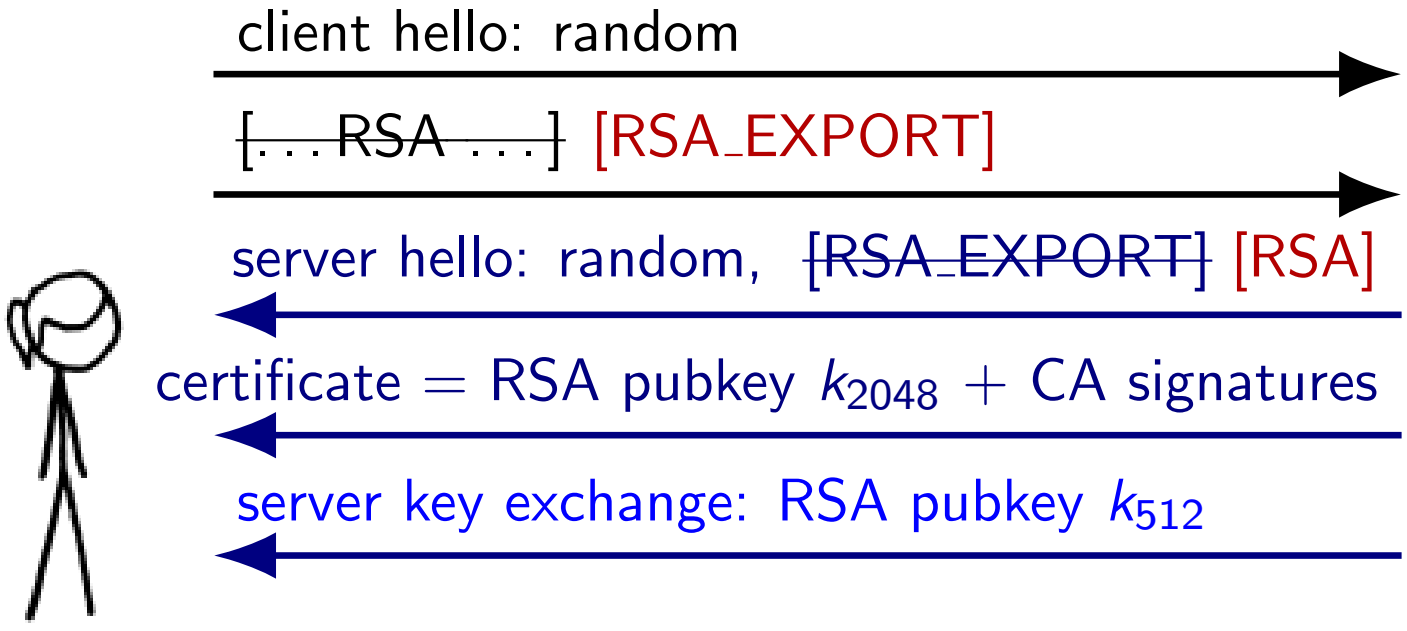
FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{mc}}$(dialog)

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

KDF(pms, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

KDF(pms, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

Enc_{k_e}(request)

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE

FREAK vulnerability in practice

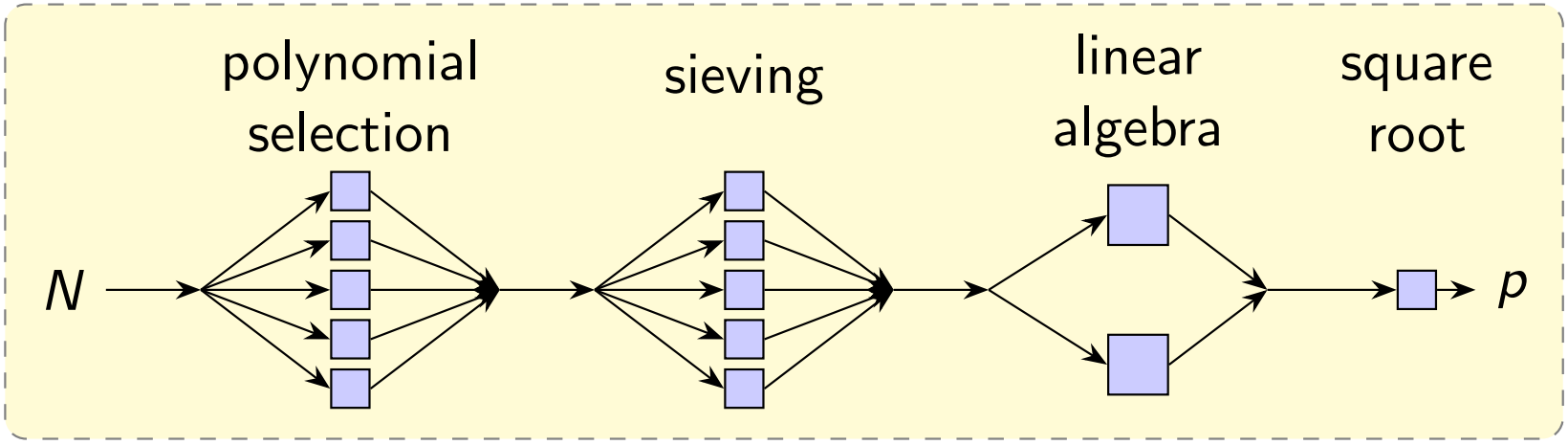
- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.
- Attacker challenge: Need to know 512-bit private key before connection times out
- Implementation shortcut: "Ephemeral" 512-bit RSA server keys generated only on application start; last for hours, days, weeks, months.

Factoring with the number field sieve

[Pollard], [Pomerance], [Lenstra,Lenstra]

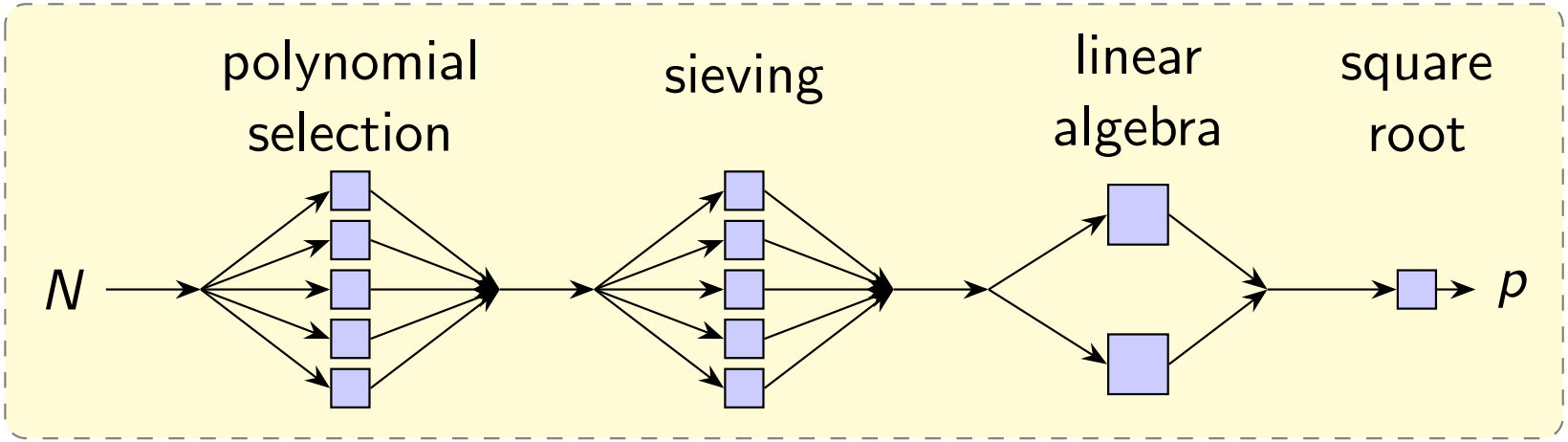


Algorithm

Motivation: Find a, b with $a^2 \equiv b^2 \bmod N$ and $\gcd(a+b, N)$ or $\gcd(a-b, N)$ nontrivial.

1. **Polynomial selection** Find a good choice of number field K.
2. **Relation finding** Factor elements over $\mathcal{O}_K$ and over $\mathbb{Z}$.
3. **Linear algebra** Find a square in $\mathcal{O}_K$ and a square in $\mathbb{Z}$.
4. **Square roots** Take square roots, map into $\mathbb{Z}$, and hope we find a factor.

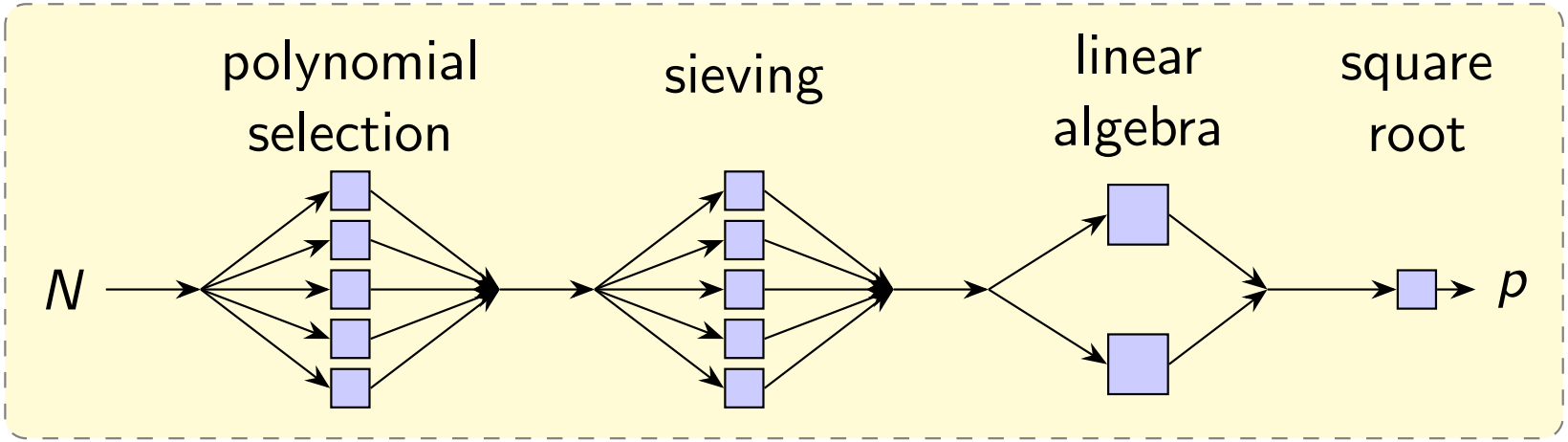
How long does it take to factor integers?



Answer 1:

$$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$$

How long does it take to factor integers?



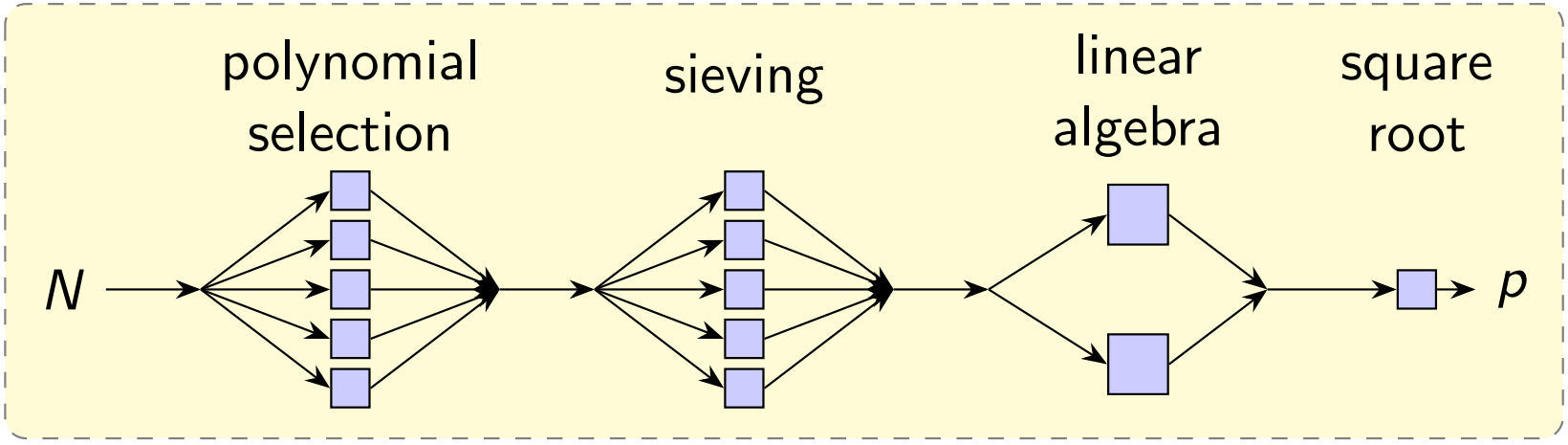
Answer 1:

$$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$$

Answer 2:

- In 1999, 512-bit RSA in 7 months and hundreds of computers. [Cavallar et al.]
- In 2009, 768-bit RSA in 2.5 calendar years. [Kleinjung et al.]

Factoring 512-bit RSA with CADO-NFS

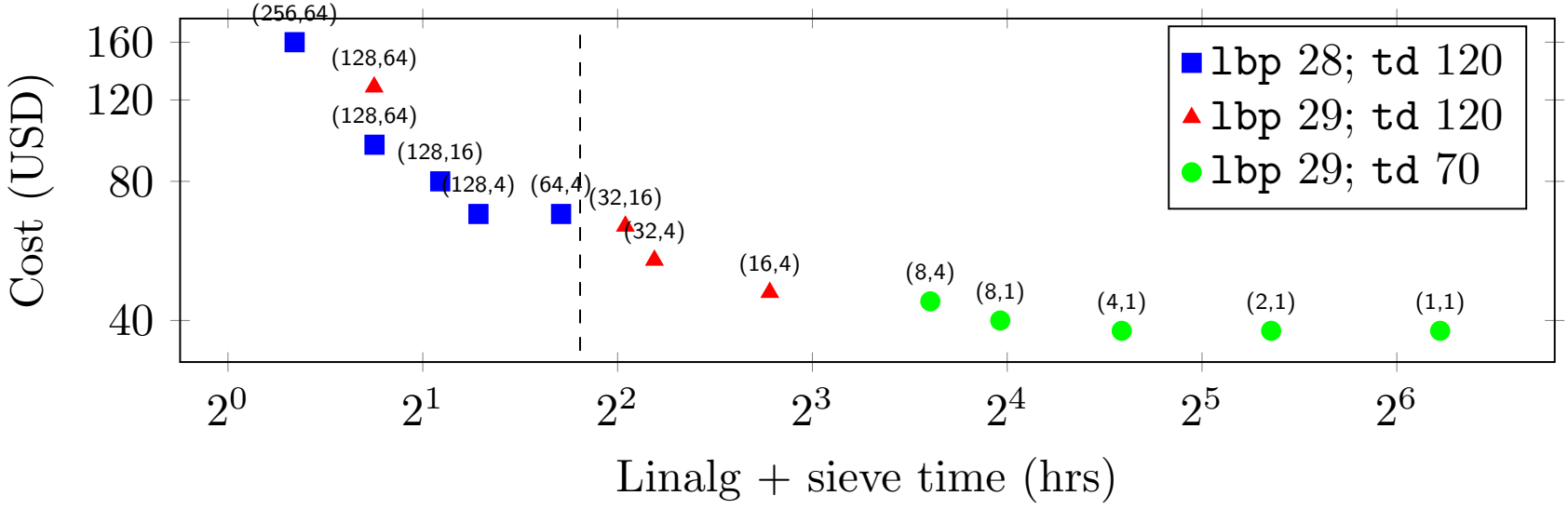


Answer 3:

	polysel	sieving	linalg	sqrt
	2400 cores		36 cores	36 cores
RSA-512	10 mins	2.3 hours	3 hours	10 mins

Factoring 512-bit RSA with CADO-NFS

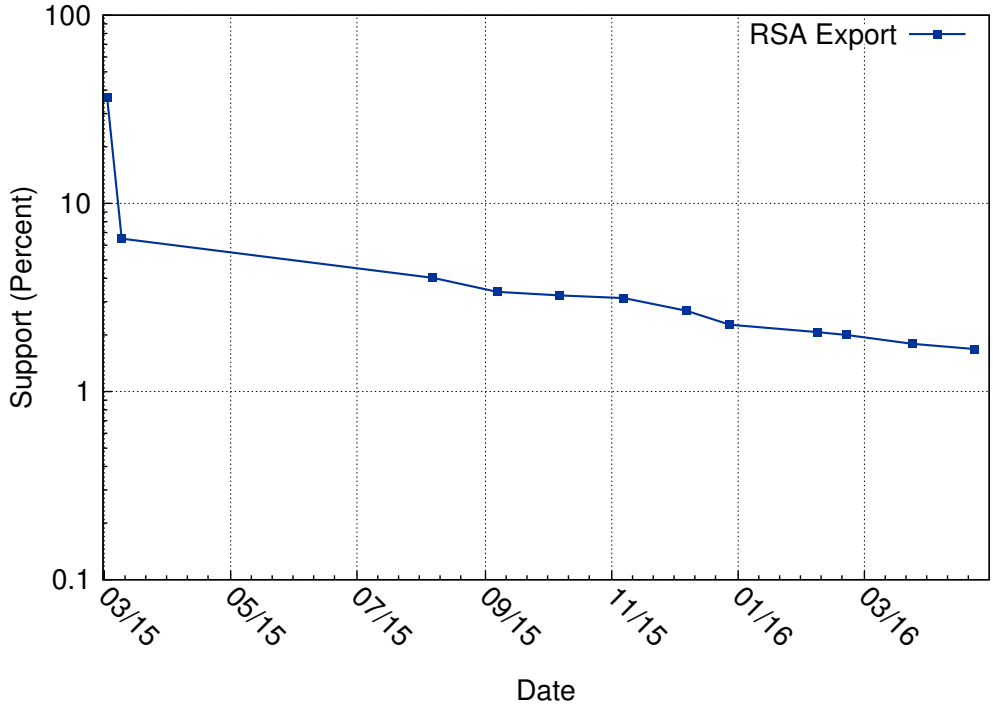
Answer 4:



Factoring as a Service Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. *FC 2016.* `seclab.upenn.edu/projects/faas/`

FREAK mitigation

- All major browsers pushed bug fixes.
- Server operators encouraged to disable export cipher suites.



But still enabled for about 2% of trusted sites today.

The Logjam attack

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 *CCS 2015* `weakdh.org`

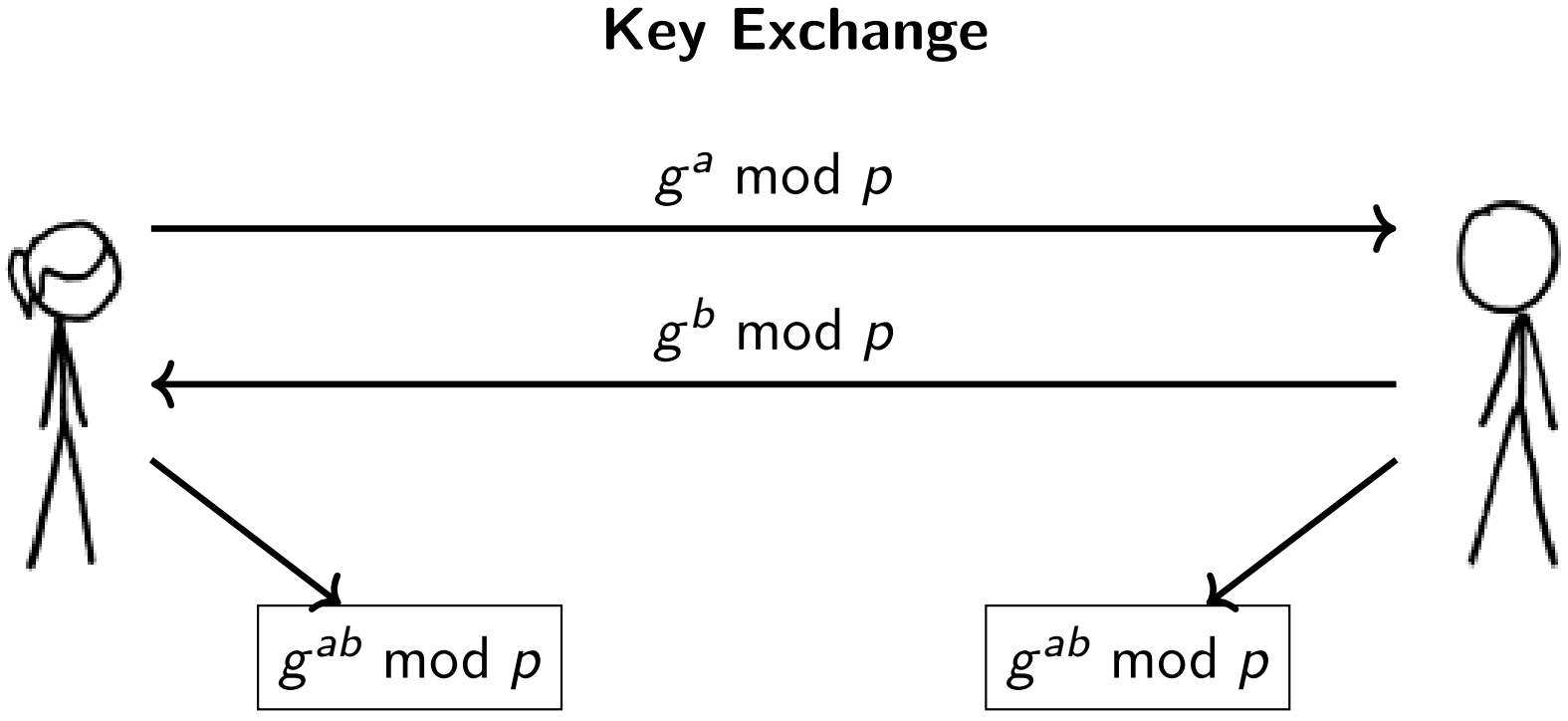
Textbook (Finite-Field) Diffie-Hellman

[Diffie Hellman 1976]

Public Parameters

p a prime (so $\mathbb{F}_p^*$ is a cyclic group)

$g < p$ group generator (often 2 or 5)



Diffie-Hellman cryptanalysis and computational problems

Discrete Log

Problem: Given g^a, compute a.

- Solving this problem permits attacker to compute shared key by computing a and raising $(g^b)^a$.
- Discrete log is in NP and coNP $\rightarrow$ not NP-complete (unless P=NP or similar).

Diffie-Hellman problem

Problem: Given g^a, g^b, compute g^{ab}.

- Exactly problem of computing shared key from public information.
- Reduces to discrete log in some cases:
- (Computational) Diffie-Hellman assumption: This problem is hard in general.

"Perfect Forward Secrecy"

"Sites that use perfect forward secrecy can provide better security to users in cases where the encrypted data is being monitored and recorded by a third party."

"With Perfect Forward Secrecy, anyone possessing the private key and a wiretap of Internet activity can decrypt nothing."

"Ideally the DH group would match or exceed the RSA key size but 1024-bit DHE is arguably better than straight 2048-bit RSA so you can get away with that if you want to."

"But in practical terms the risk of private key theft, for a non-ephemeral key, dwarfs out any cryptanalytic risk for any RSA or DH of 1024 bits or more; in that sense, PFS is a must-have and DHE with a 1024-bit DH key is much safer than RSA-based cipher suites, regardless of the RSA key size."

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\text{Auth}_{k_{m_c}}(\text{dialog})$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\text{Auth}_{k_{m_c}}$(dialog)

server finished: $\text{Auth}_{k_{m_s}}$(dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\text{Auth}_{k_{m_c}}$(dialog)

server finished: $\text{Auth}_{k_{m_s}}$(dialog)

Enc_{k_e}(request)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Question: How do you selectively weaken a protocol based on Diffie-Hellman?

Question: How do you selectively weaken a protocol based on Diffie-Hellman?

Export answer: Optionally use a smaller prime.

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

server hello: server random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

server hello: server random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

server hello: server random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\mathrm{Auth}_{k_{m_c}}$(dialog)

server finished: $\mathrm{Auth}_{k_{m_s}}$(dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

server hello: server random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\mathrm{Auth}_{k_{m_c}}$(dialog)

server finished: $\mathrm{Auth}_{k_{m_s}}$(dialog)

Enc_{k_e}(request)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Diffie-Hellman export cipher suites in TLS

```
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_Anon_EXPORT_WITH_RC4_40_MD5
TLS_DH_Anon_EXPORT_WITH_DES40_CBC_SHA
```

April 2015: 8.4% of Alexa top 1M HTTPS support `DHE_EXPORT`.

Totally insecure, but no modern client would negotiate export ciphers. ... right?

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

[. . . DHE . . .]

Logjam: Active downgrade attack to export Diffie-Hellman

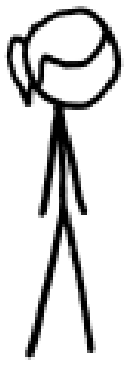
Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

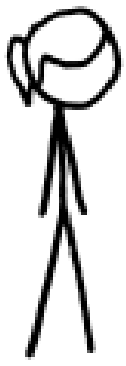
server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

$\mathrm{KDF}(g^{ab}, \mathrm{randoms}) \rightarrow k_{m_c}, k_{m_s}, k_e$

$\mathrm{KDF}(g^{ab}, \mathrm{randoms}) \rightarrow k_{m_c}, k_{m_s}, k_e$

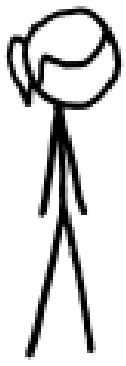
Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]



certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

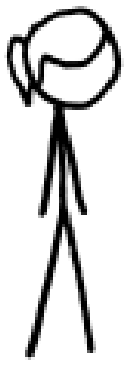
Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]



certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

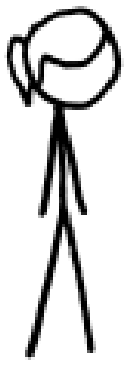
client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{m_c}}$(dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

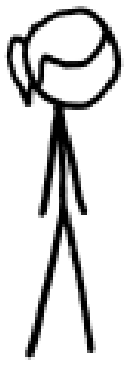
server finished: $\mathrm{Auth}_{k_{m_s}}$(modified dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

Enc_{k_e}(request)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

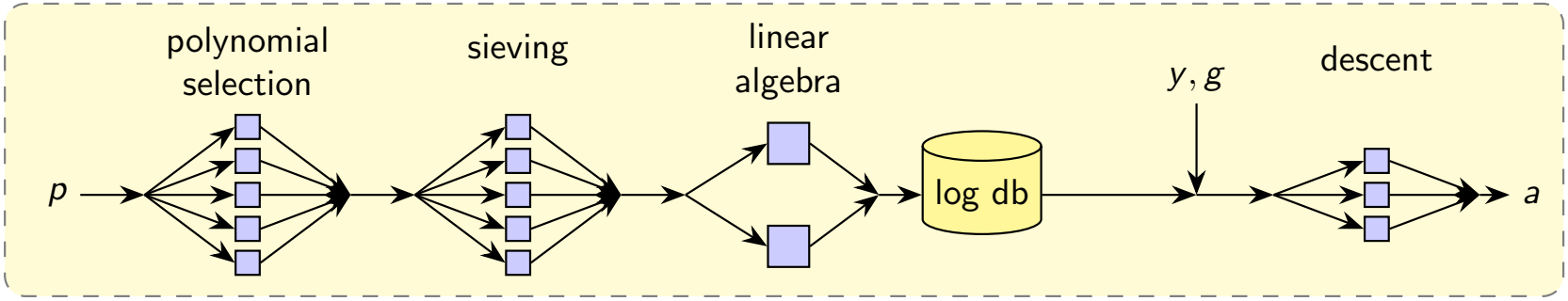
Carrying out the Diffie-Hellman export downgrade attack

1. Attacker man-in-the-middles connection, changing messages as necessary.
2. Attacker computes discrete log of g^a or g^b to learn session keys.
3. Attacker uses session keys to forge client, server finished messages.

- Attacker challenge: compute client or server ephemeral Diffie-Hellman secrets before connection times out
- For export Diffie-Hellman, most servers actually generate per-connection secrets.

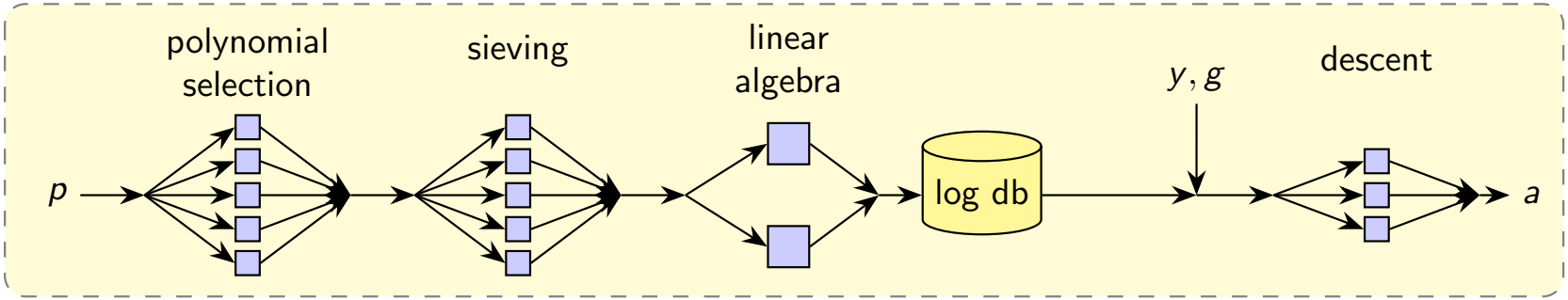
Number field sieve discrete log algorithm

[Gordon], [Joux, Lercier], [Semaev]



1. **Polynomial selection**: Find a good choice of number field K.
2. **Relation collection**: Factor elements over $\mathcal{O}_K$ and over $\mathbb{Z}$.
3. **Linear algebra** Once there are enough relations, solve for logs of small elements.
4. **Individual log** "Descent" Try to write target t as sum of logs in known database.

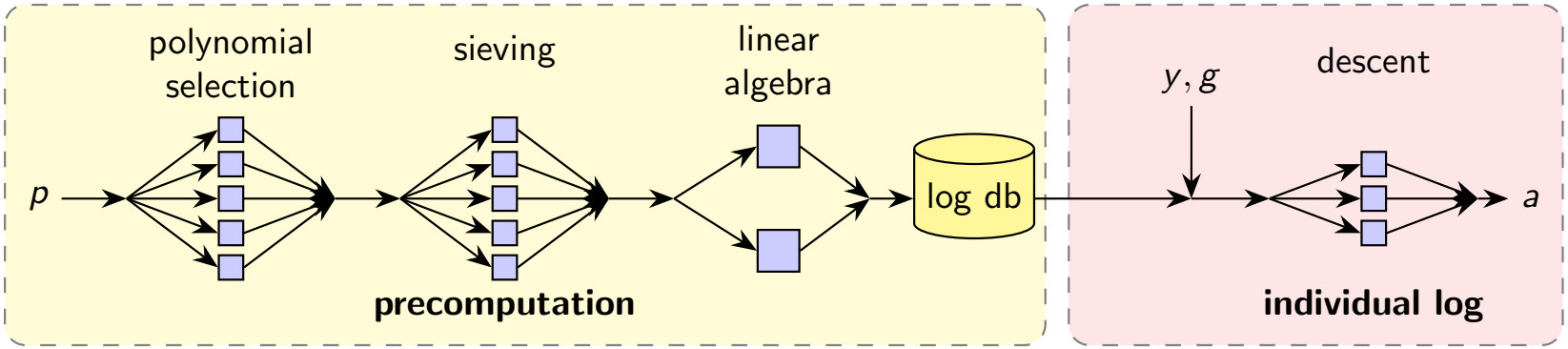
How long does it take to compute discrete logs?



Answer 1:

$$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$$

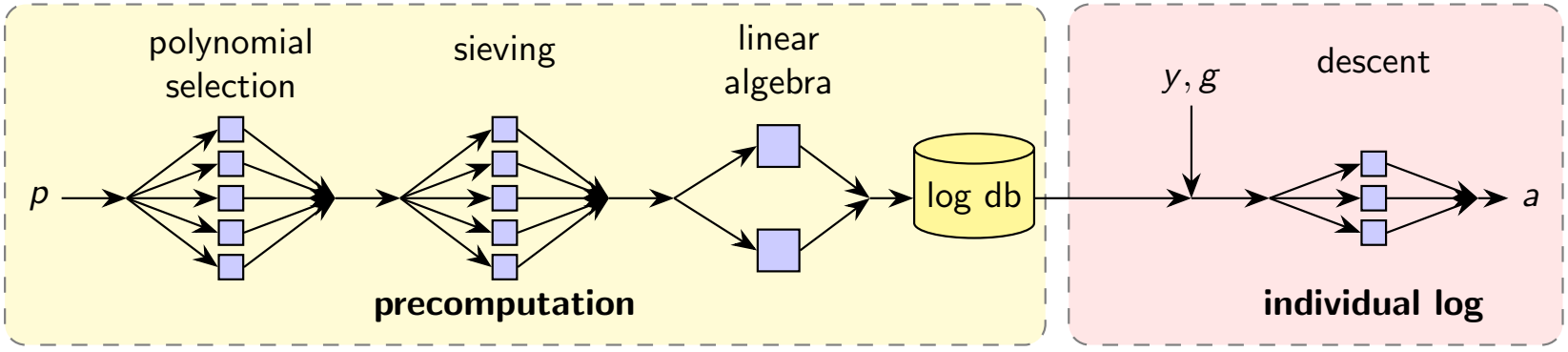
How long does it take to compute discrete logs?



Answer 1:

$$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$$

How long does it take to compute discrete logs?

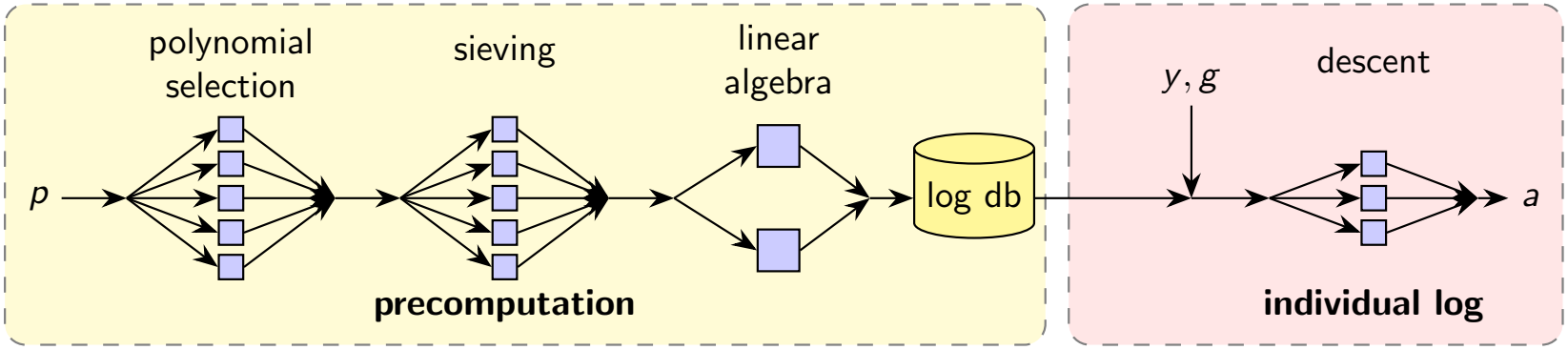


Answer 1:

$L(1/3, 1.232)$

$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$

How long does it take to compute discrete logs?



Answer 1:

$L(1/3, 1.232)$

$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log\log N)^{2/3})$

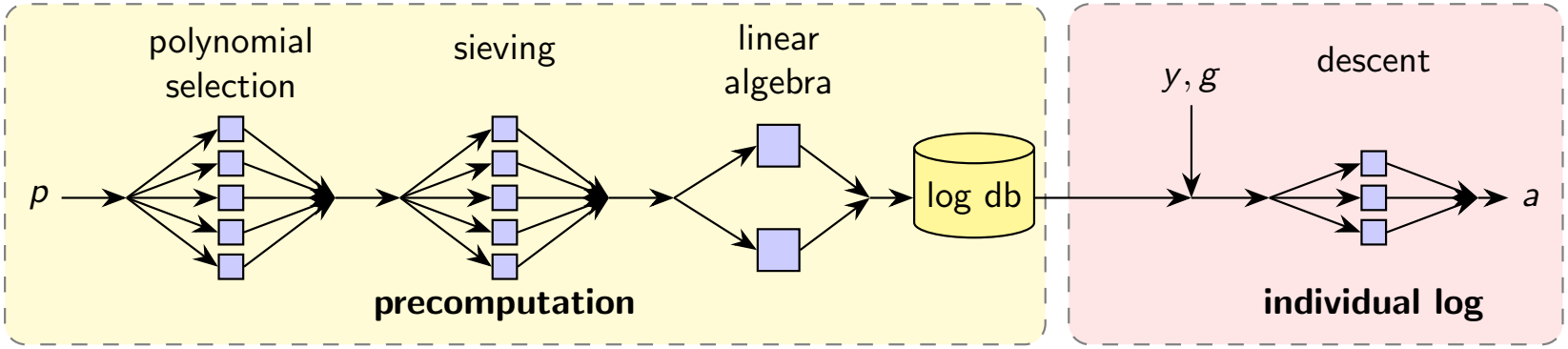
Answer 2:

In 2005, 431-bit discrete log in 3 weeks. [Joux, Lercier]

In 2007, 530-bit discrete log. [Kleinjung]

In 2014, 596-bit discrete log. [Bouvier et al.]

How long does it take to compute discrete logs?



Answer 3:

	polysel	sieving	linalg	descent
	2000-3000 cores		288 cores	36 cores
DH-512	3 hours	15 hours	120 hours	70 seconds

Precomputation can be done once and reused for many individual logs!

Implementing Logjam

Parameters hard-coded in implementations or built into standards.

97% of `DHE_EXPORT` hosts choose one of three 512-bit primes.

Hosts	Source	Year	Bits
80%	Apache 2.2	2005	512
13%	mod_ssl 2.3.0	1999	512
4%	JDK	2003	512

- Carried out precomputation for common primes.
- After 1 week precomputation, median individual log time 70s.
- Logjam and our precomputations can be used to break connections to 8% of the HTTPS top 1M sites!

CA Services Test @CAServicesBot · 8s

@DLogBot What's up bot?

1

CA Services

@DLogBot

Following

@CAServicesBot Thank you for using the CA Services discrete logarithm bot. Your request should be <group (a/m/o)> <ephemeral key in hex>

3:58 PM - 12 Aug 2015

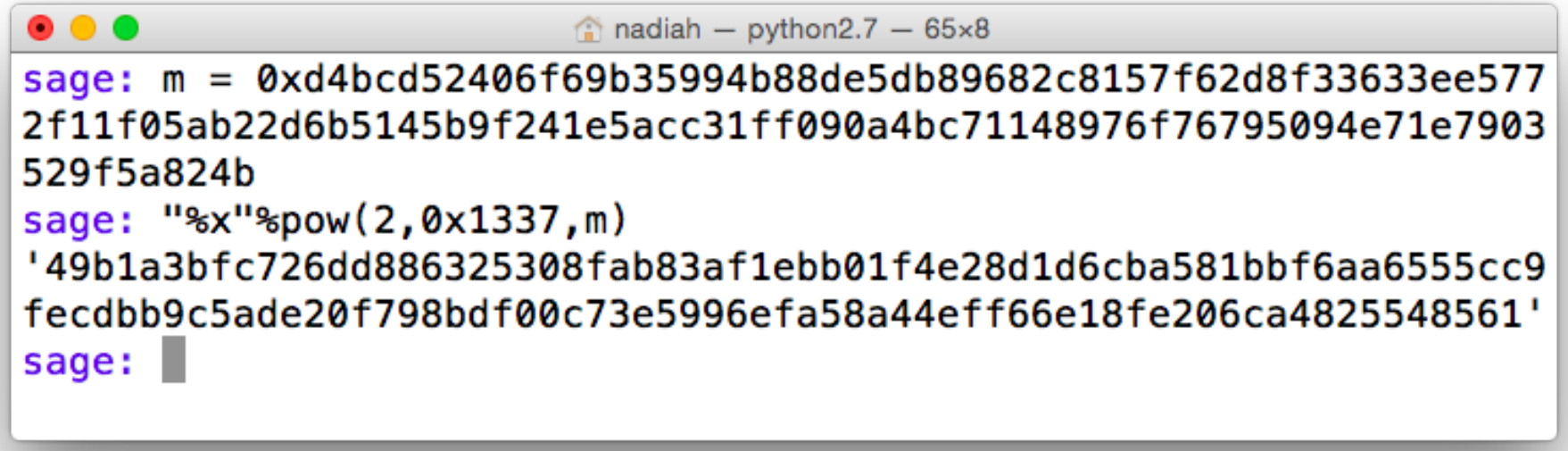
Reply to @DLogBot

g = 2

apache:
9fdb8b8a004544f0045f1737d0ba2e0b274cdf1a9f588218fb43
5316a16e374171fd19d8d8f37c39bf863fd60e3e300680a3030c
6e4c3757d08f70e6aa871033

openssl:
da583c16d9852289d0e4af756f4cca92dd4be533b804fb0fed94e
f9c8a4403ed574650d36999db29d776276ba2d3d412e218f4dd1e
084cf6d8003e7c4774e833

mod_ssl:
d4bcd52406f69b35994b88de5db89682c8157f62d8f33633ee577
2f11f05ab22d6b5145b9f241e5acc31ff090a4bc71148976f7679
5094e71e7903529f5a824b


nadiah — python2.7 — 65×8
sage: m = 0xd4bcd52406f69b35994b88de5db89682c8157f62d8f33633ee5772f11f05ab22d6b5145b9f241e5acc31ff090a4bc71148976f76795094e71e7903529f5a824b
sage: "%x"%pow(2,0x1337,m)
'49b1a3bfc726dd886325308fab83af1ebb01f4e28d1d6cba581bbf6aa6555cc9fecdbb9c5ade20f798bdf00c73e5996efa58a44eff66e18fe206ca4825548561'
sage:

Tweet to CA Services

@DLogBot m
49b1a3bfc726dd886325308fab83af1ebb01f4e28d1d6cba581bbf6aa6555cc9fecdbb9c5ade2
0f798bdf00c73e5996efa58a44eff66e18fe206ca4825548561

Add photo Location disabled 1 Tweet

View conversation

CA Services @DLogBot · 5m
@CAServicesBot
9fa918b3beea9b3a1d564d5656b0f2f20d4f05062eed6a9eecf48bc2119
e0d41be1bd418b29e4feff5600645f5fd80bcd71190bbe6cf6e592be513
9219414

View conversation

Who to follow

Joe Fi

Maryl

Kevin

Find friends

Trends · Chang

```
900755693880224784181826846280507 89
Wed Aug 12 18:45:33 2015 Deduced log of (470366623, 338113733, 1) from rel: 13237548435244169244143100182094873380360838654192799064225833828132022218196575848107253557044684872978237019555315126021801420268766874662578866567740811
Wed Aug 12 18:45:33 2015 Deduced log of (25962401291, 20249792848, 0) from rel: 505633307374159604150495753286374600982496989595759740618770110704480701624053426981957192559057945796090106755289534153710885079574744434237737252424012
Wed Aug 12 18:45:33 2015 Deduced log of (152250701, 93107759, 0) from rel: 368622813850158283152826798929907412284902377185469178829870371289144771591876492349659700969350065276910486022078530561941461460445126822792313541759652 2
Wed Aug 12 18:45:33 2015 Deduced log of (143560981, 107192704, 1) from rel: 307860298231873673521041039296250709974237070003579136705638665216739899169781455078181073498093656533016603801659935296931860439561657504329645695298987 5
Wed Aug 12 18:45:33 2015 Deduced log of (278147959, 4410170, 1) from rel: 250568495748563924992223084465769388412691186709473708889010485389673852742627612820328323174655027075651009937568594328296740001105522457321813526033020 8
Wed Aug 12 18:45:33 2015 Deduced log of (577776097, 143348514, 1) from rel: 442338900520785256843955526485021247124215862664804946530253545141243702700937987405857426583528963422304799351630803791913894065413730465443510800769091 6
Wed Aug 12 18:45:33 2015 Deduced log of (418914989, 316721060, 0) from rel: 367207948543056444750000809717878390413543347607219059542512916467040252344925831757084814255582491568962667272894312894745432354830030222594946416351771128
Wed Aug 12 18:45:33 2015 Deduced log of (338071753, 301777175, 1) from rel: 107520094080132598180876691823882310946126491164966924787589553472432468656483029744406591626920263170944975959228432814632855732906875574159648657015326 9
Wed Aug 12 18:45:33 2015 Deduced log of (334373141, 196775704, 0) from rel: 295924098567118682024550377212916862876453441769461426617303202625502622200306437293865652865355209952938669043254101956488326928215586243832945368451306 8
Wed Aug 12 18:45:33 2015 Deduced log of (85126349747, 78086389814, 1) from rel: 272008891507334686458442924203233155691323232288533240018774772348710672437432344405421003034049598156144493564929239372773533486332215126820403976503407
Wed Aug 12 18:45:33 2015 Deduced log of (61692766831901, 45737121448216, 0) from rel: 950385701946810568492835855519755175335271777937141287749717580863420767092852870952282318514359529666745662554115535892794666946074238694583128167593602 3
Wed Aug 12 18:45:33 2015 Deduced log of (206206811, 193918773, 1) from rel: 150129281584625653670608607473569366162828241135446571302979567882130921962371540905953102043599924796633201952837388941979655233459332812014626766068665 66
Wed Aug 12 18:45:33 2015 Deduced log of (226880903, 205068533, 1) from rel: 536845220976247216070823275266198080149544131379113941538069691920397796698875020561335789576215785486392872542467451478977409964811844057396299029579 51
Wed Aug 12 18:45:33 2015 Deduced log of (292902821, 134190188, 1) from rel: 439445138364582159970955146631236342487943352438465485528919566060271737879221218557114979606253028169269932103194756547132565957464751660244704990870 5083
Wed Aug 12 18:45:33 2015 Deduced log of (1307702399, 951504975, 0) from rel: 388412984539454708054236165551175352347925108926611462668495683292715649026161881839025602408585455677091772199785578032896317573836498043317135746913601 0
Wed Aug 12 18:45:33 2015 Deduced log of (202009439, 132696313, 1) from rel: 586746728022223145407986515759301518747360666540921337722240732928530330678243336945997816779047558926178822228470304876293629942892685459127281244685422
Wed Aug 12 18:45:33 2015 Deduced log of (277410803, 9791152, 1) from rel: 225900191667372239717508053757873005106976563413582561173459642209893512269377176143983896388166851314560890572508498540819037064719778284596654350279821 3
Wed Aug 12 18:45:33 2015 Deduced log of (5077939123, 4652063990, 1) from rel: 4700476107471785387546249836018003783883820730918635794744673497035689116025731296286088760225494958896998096151611592457429278405080150250145357601977064
Wed Aug 12 18:45:33 2015 Deduced log of (570419228219, 397375869245, 0) from rel: 3255654853015642854084419126761488294889128335840284315033305299124899394204172237584087433608546207517017017002433773924631799147022786959354304587524964
Wed Aug 12 18:45:33 2015 Deduced log of (149460671, 53992188, 1) from rel: 2651811832073407132611177858759137310517057082625851781864665609907080436715896016549107955625430256337426585266782281625402584417872997921002003028988191
Wed Aug 12 18:45:33 2015 Deduced log of (1004549233, 320473819, 1) from rel: 4364592475872277799643025982830330362613984403493365342236185699223330907782331434167819646594219292852478453423087536704765635181511493534233116851018079
Wed Aug 12 18:45:33 2015 Deduced log of (500400001, 21453686, 0) from rel: 1824485543248714996126611031082238658686942161900511996641160312138508104939395699652788432709468236780128943689748040801275264279266440000948459876830721
Wed Aug 12 18:45:33 2015 # p=111419736167993051826721259538215396217898638646520821894844183837557977265003942122273143385094102252406211275129135279721296280500639181169859831840446 19
Wed Aug 12 18:45:33 2015 # ell=557098680839965259133606297691076981089493193232604109474220919187789886325019710611365716925470511262031056375645676398606481402503195905849299159202230 9
Wed Aug 12 18:45:33 2015 log(2)=47197552632340568686371246544797334661553247954679670451641942805650991309645074077087810506576181574948593419547569918183935151167199904147667669979273 05
Wed Aug 12 18:45:33 2015 log(3)=20495757144121934877222047902663003624011491743961240029434474987370825650259443439987285179799353725777366586067257147915224641099414699956801923466367 33
Wed Aug 12 18:45:33 2015 log(7)=56085170553927017859636655668265525368994714302698459590550178723216657331070298461703411206941866056867006913261115179252433760356477128350339490874729626
Wed Aug 12 18:45:33 2015 # target=385966297662854851100984432444339538413671929321908207345189346625768237659353266583704409102443785749659034063226219416703000988881604865261883330006358 5
Wed Aug 12 18:45:33 2015 log(target)=21741092469733887286417505986311180188613069043166533718859635445180620508405973438845639004675124283789839022943072247456208168374594534974308988474516 92
Wed Aug 12 18:45:33 2015 Final consistency check ok!
```

CA Services Test @CAServicesBot · 3m

@DLogBot m
49b1a3bfc726dd886325308fab83af1ebb01f4e28d1d6cba581bbf6aa6555cc9fe
cdbb9c5ade20f798bdf00c73e5996efa58a44eff66e18fe206ca4825548561

1

CA Services
@DLogBot

Following

@CAServicesBot 1337

:45 PM - 12 Aug 2015

Reply to @DLogBot

Daniel J. Bernstein @hashbreaker · Aug 12

@DLogBot m
5a2790dac75a8f9456da6f57ff117b1078f3a1472810a7bfdecb61ea8e43ce8fa16b
b019acf670ae98ed1cf9064b5a3f96fa5348ea5af7b949e10bf56b18f39f

4 5

CA Services
@DLogBot

Follow

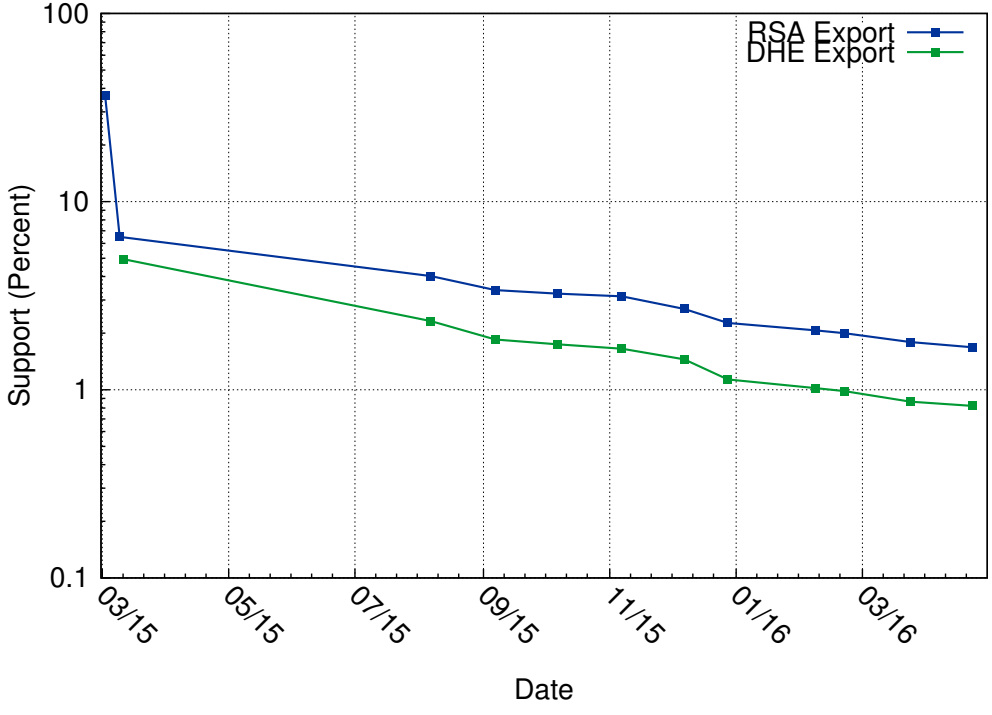
.@hashbreaker
bada55ecc00031415926535897932 3

RETWEETS 8 FAVORITES 10

5:19 PM - 12 Aug 2015

Logjam mitigation

- Server operators encouraged to disable export cipher suites.



- Major browsers have raised minimum DH lengths:
  IE, Chrome, Firefox to 1024 bits; Safari to 768.

- TLS 1.3 draft includes anti-downgrade flag in client random.

The legacy of export-grade cryptography in the 21st century

Nadia Heninger and J. Alex Halderman

University of Pennsylvania University of Michigan

June 9, 2016

Review of Part 1

- 1990s: U.S. cryptography regulations limit strength of RSA, Diffie-Hellman, and symmetric ciphers in exported products.
- 1990s: Netscape develops SSL, complies with regulations by supporting weakened export-grade cryptography.

. . . *20 years later* . . .

- March 2015: **FREAK attack**
  Modern, full-strength TLS connections can be downgraded to 512-bit export-grade RSA; attacker can factor to decrypt session data. 10% of popular HTTPS sites vulnerable.
- May 2015: **Logjam attack**
  Modern, full-strength TLS connections can be downgraded to 512-bit export-grade Diffie-Hellman; attacker can take discrete log to decrypt session data. 8% of popular HTTPS sites vulnerable.

First, a digression...

Things we learned about Diffie-Hellman in practice

Logjam attack:

Anyone can use backdoors from '90s crypto war to attack modern browsers.

First, a digression...

Things we learned about Diffie-Hellman in practice

Logjam attack:
Anyone can use backdoors from '90s crypto war to attack modern browsers.

Mass surveillance:
Governments can exploit 1024-bit discrete log for wide-scale passive decryption.

Is breaking 1024-bit Diffie-Hellman within reach of governments?

	Sieving			Linear Algebra		Descent
	I	lpb	core-years	rows	core-years	core-time
RSA-512	14	29	0.5	4.3M	0.33	
DH-512	15	27	2.5	2.1M	7.7	10 mins
RSA-768	16	37	800	250M	100	
DH-768	17	35	8,000	150M	28,500	2 days
RSA-1024	18	42	1,000,000	8.7B	120,000	
DH-1024	19	40	10,000,000	5.2B	35,000,000	30 days

Is breaking 1024-bit Diffie-Hellman within reach of governments?

	Sieving			Linear Algebra		Descent
	I	lpb	core-years	rows	core-years	core-time
RSA-512	14	29	0.5	4.3M	0.33	
DH-512	15	27	2.5	2.1M	7.7	10 mins
RSA-768	16	37	800	250M	100	
DH-768	17	35	8,000	150M	28,500	2 days
RSA-1024	18	42	1,000,000	8.7B	120,000	
DH-1024	19	40	10,000,000	5.2B	35,000,000	30 days

- Special-purpose hardware $\rightarrow \approx 80\times$ speedup.
  (Research problem: Make rigorous!)
- $\approx$\$100Ms machine precomputes for one 1024-bit p every year
- Then, individual logs can be computed in close to real time

James Bamford, 2012, Wired

According to another top official also involved with the program, the NSA made an enormous breakthrough several years ago in its ability to cryptanalyze, or break, unfathomably complex encryption systems employed by not only governments around the world but also many average computer users in the US. The upshot, according to this official: "Everybody's a target; everybody with communication is a target."

[...]

The breakthrough was enormous, says the former official, and soon afterward the agency pulled the shade down tight on the project, even within the intelligence community and Congress. "Only the chairman and vice chairman and the two staff directors of each intelligence committee were told about it," he says. The reason? "They were thinking that this computing breakthrough was going to give them the ability to crack current public encryption."

2013 NSA "Black Budget"

"Also, we are investing in groundbreaking cryptanalytic capabilities to defeat adversarial cryptography and exploit internet traffic."

This Exhibit is SECRET//NOFORN

Program	Expenditure Center	Project	FY 2011	FY 2012	FY 2013	FY 2012 - FY 2013 Change
	Computer Network Operations	Data Acquisition and Cover Support	56,949	100,987	117,605	16,618
		GENIE	615,177	636,175	651,743	15,568
		SIGINT Enabling	298,613	275,376	254,943	-20,433
	Computer Network Operations Total		**970,739**	**1,012,538**	**1,024,291**	**11,753**
	Cryptanalysis & Exploitation Services	Analysis of Target Systems	39,429	35,128	34,321	-807
		Cryptanalytic IT Systems	130,012	136,797	247,121	110,324
		Cyber Cryptanalysis	181,834	110,673	115,300	4,627
		Exploitation Solutions	90,024	59,915	58,308	-1,607
		Microelectronics	64,603	61,672	45,886	-15,786

*numbers in thousands

Parameter reuse for 1024-bit Diffie-Hellman

- Precomputation for a single 1024-bit prime allows passive decryption of connections to 66% of VPN servers and 26% of SSH servers.

  (Oakley Group 2)

- Precomputation for a second common 1024-bit prime allows passive decryption for 18% of top 1M HTTPS domains.

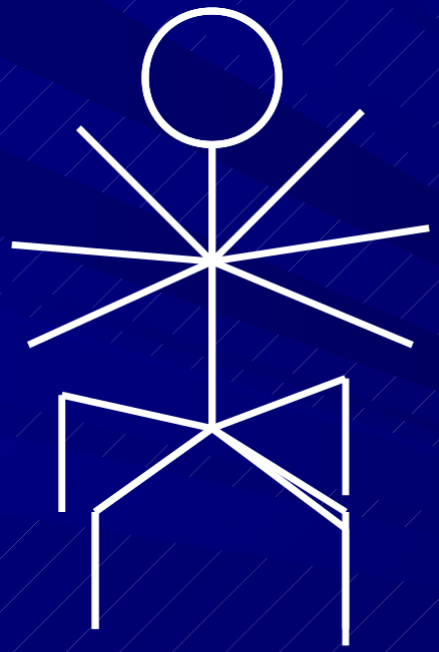
  (Apache 2.2)

4. Communicate Results

Can we decrypt the VPN traffic?

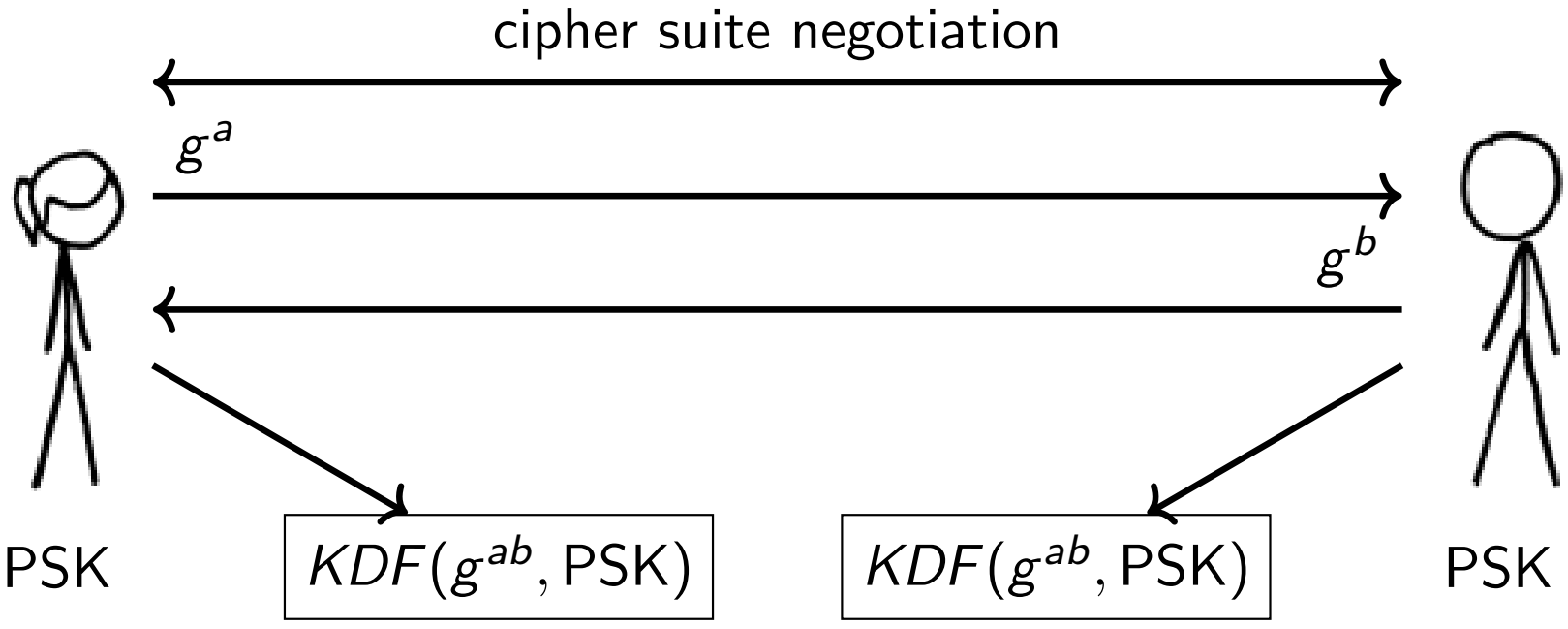
- If the answer is “No” then explain how to turn it into a “YES!”
- If the answer is “YES!” then…

Happy Dance!!



IKE Key Exchange for IPsec VPNs

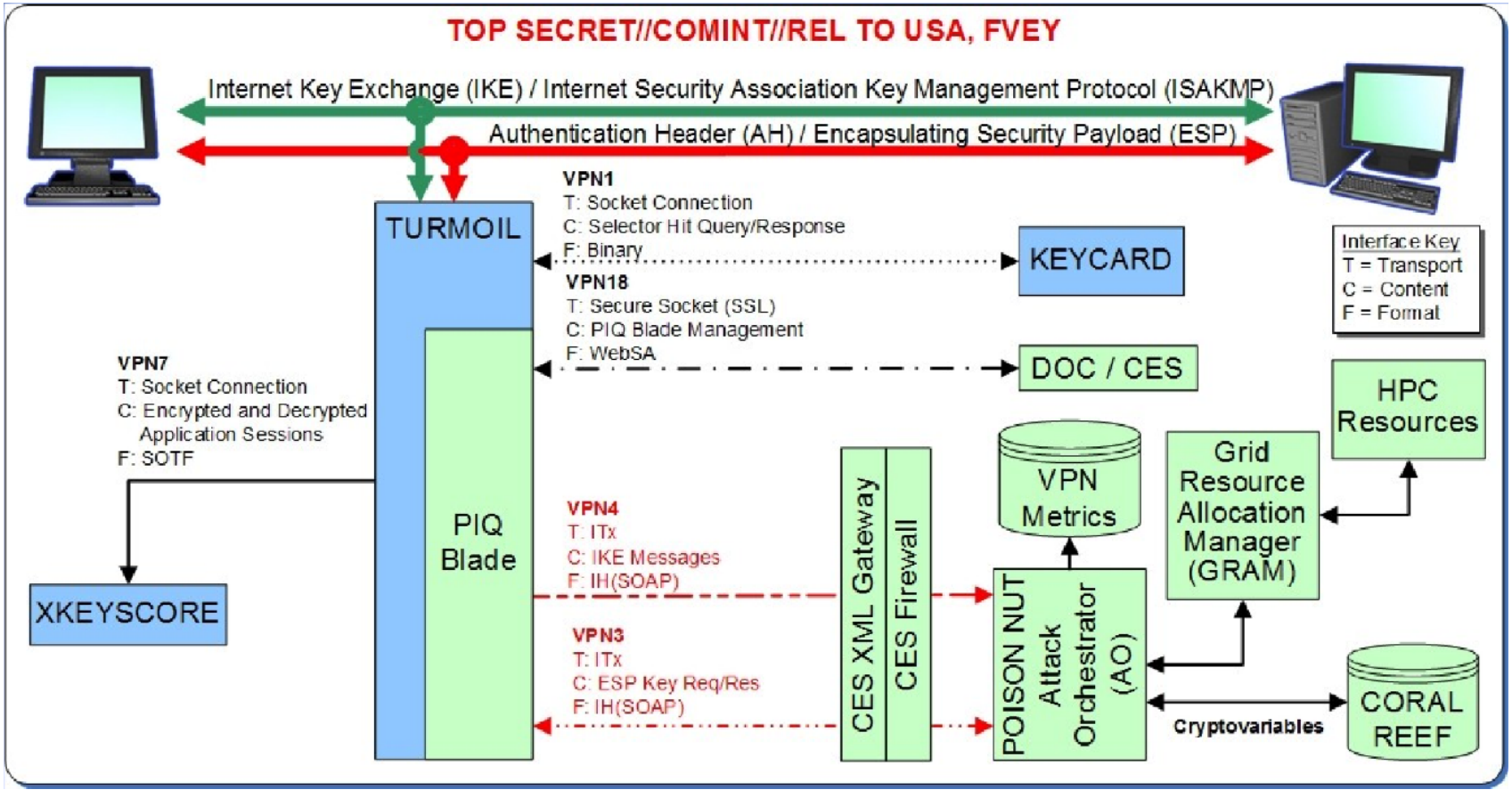
IKE chooses Diffie-Hellman parameters from standardized set.



Turn that Frown Upside Down! From “No” to “YES!”

- Depends on why we couldn’t decrypt it
- Find Pre-Shared Key
- Locate complete paired collect
- Locate both IKE and ESP traffic
- Have collection sites do surveys for the IP’s
- Find better quality collect with rich metadata

NSA VPN Attack Orchestration



NSA's on-demand IKE decryption requires:

- Known pre-shared key.
- Both sides of IKE handshake.
- Both IKE handshake and ESP traffic.
- IKE+ESP data is sent to HPC resources.

Discrete log decryption would require:

- Known pre-shared key.
- Both sides of IKE handshake.
- Both IKE handshake and ESP traffic.
- IKE data sent to HPC resources.

A well-designed "implant" would have fewer requirements.

	Vulnerable servers, if attacker can precompute for		
	all 512-bit p	one 1024-bit p	ten 1024-bit p
HTTPS Top 1M MITM	45K (8.4%)	205K (37.1%)	309K (56.1%)
HTTPS Top 1M	118 (0.0%)	98.5K (17.9%)	132K (24.0%)
HTTPS Trusted MITM	489K (3.4%)	1.84M (12.8%)	3.41M (23.8%)
HTTPS Trusted	1K (0.0%)	939K (6.56%)	1.43M (10.0%)
IKEv1 IPv4	–	1.69M (66.1%)	1.69M (66.1%)
IKEv2 IPv4	–	726K (63.9%)	726K (63.9%)
SSH IPv4	–	3.6M (25.7%)	3.6M (25.7%)

Diffie-Hellman Attacks and Mitigations

Logjam attack:

Anyone can use backdoors from '90s crypto war to pwn modern browsers.

Mitigations:

- Major browsers raised minimum DH lengths.
- TLS 1.3 draft anti-downgrade mechanism.
- Recommendation: Don't backdoor crypto!

Mass surveillance:

Governments can exploit 1024-bit discrete log for wide-scale passive decryption.

Mitigations:

- Move to elliptic curve cryptography
- If ECC isn't an option, use $\geq$ 2048-bit primes.
- If 2048-bit primes aren't an option, generate a fresh 1024-bit prime.

Continuing from Part 1

- 1990s: U.S. cryptography regulations limit strength of RSA, Diffie-Hellman, and symmetric ciphers in exported products.
- 1990s: Netscape develops SSL, complies with regulations by supporting weakened export-grade cryptography.

. . . *20 years later* . . .

- March 2015: **FREAK attack**
  Modern, full-strength TLS connections can be downgraded to 512-bit export-grade RSA; attacker can factor to decrypt session data. 10% of popular HTTPS sites vulnerable.
- May 2015: **Logjam attack**
  Modern, full-strength TLS connections can be downgraded to 512-bit export-grade Diffie-Hellman; attacker can take discrete log to decrypt session data. 8% of popular HTTPS sites vulnerable.
- What about export-grade symmetric ciphers?

The DROWN attack

DROWN: Breaking TLS using SSLv2

Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt

To appear in *USENIX Security 2016.* `https://drownattack.com`

A brief history of SSL/TLS

SSL 1.0	Terribly insecure; never released.
SSL 2.0	Released 1995; terribly insecure.
SSL 3.0	Released 1996; considered insecure since 2014/POODLE.
TLS 1.0	Released 1999.
TLS 1.1	Released 2006.
TLS 1.2	Released 2008.
TLS 1.3	Under development.

Clients will negotiate highest supported version, so it's ok for servers to support old versions for compatibility ...right?

Question: How do you selectively weaken a protocol that uses symmetric ciphers?

Question: How do you selectively weaken a protocol that uses symmetric ciphers?

SSLv2 export answer: Optionally send all but 40 bits of secret key in public.

$mk = mk_{clear} \parallel mk_{secret}$

SSLv2 Handshake

client hello: [cipher suites], challenge

SSLv2 Handshake

client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

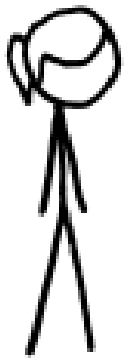
SSLv2 Handshake

client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)



KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

SSLv2 Handshake

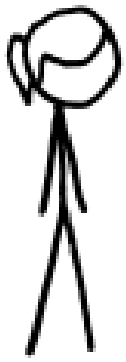
client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)

server verify: Enc_{k_e}(challenge)



KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

SSLv2 Handshake

client hello: [cipher suites], challenge

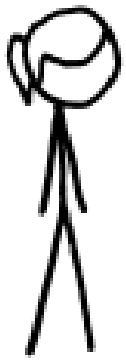
server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)

server verify: Enc_{k_e}(challenge)

client finished



KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

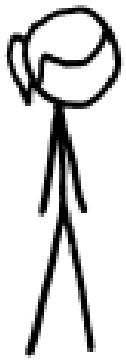
KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

SSLv2 Handshake

client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures



$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)

server verify: Enc_{k_e}(challenge)

client finished

server finished

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

SSLv2 Handshake

client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

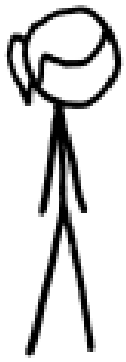
$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)

server verify: Enc_{k_e}(challenge)

client finished

server finished

Enc_{k_e}(request)



KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Notable properties of SSLv2

- Devastating MITM attacks.
- RSA key exchange only.
- `master_key` varies in size according to symmetric cipher. For TLS, premaster secret always has 48 bytes.
- Both encryption and authentication use 40-bit symmetric secret for export cipher suites. (TLS export cipher suites extract 40-bit secret for encryption from 48-byte PMS.)
- Server authenticates first. (Not well specified in spec; implementations agree.)

A garden of attacks on textbook RSA

Unpadded RSA encryption is homomorphic under multiplication. Let's have some fun!

Attack: Malleability

Given a ciphertext $c = \text{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\text{Enc}(ma) = ca^e \bmod N$ for any a.

Attack: Chosen ciphertext attack

Given a ciphertext $c = \text{Enc}(m)$ for unknown m, attacker asks for $\text{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

So in practice **must always use padding on messages**.

RSA PKCS #1 v1.5 padding

`m = 00 02 [random padding string] 00 [data]`

- Encrypter pads message, then encrypts padded message using RSA public key.
- Decrypter decrypts using RSA private key, strips off padding to recover original data.

Q: What happens if a decrypter decrypts a message and the padding isn't in correct format?

A: Throw an error?

[Bleichenbacher 1998] padding oracle attacks

- **Attack:** If no padding error, attacker learns that first two bytes of plaintext are 00 02.
- Error messages are *oracle* for first two bytes of plaintext.

Adaptive chosen ciphertext attack:

1. Attacker wishes to decrypt RSA ciphertext c.
2. Attacker queries oracle on $s^e c \bmod N$ for random values of s.
3. Attacker successively narrows range of possible m until unique answer remains . . . eventually. *"Million message attack."*

Mitigation: ~~Use OAEP.~~ Implementations don't reveal errors to attacker; proceed with protocol using fake random plaintext.

Using an SSLv2 server as a Bleichenbacher oracle

- Attacker wishes to learn whether test ciphertext c has valid PKCS padding.
- Server sends `ServerVerify` message before client authenticates!
- If padding incorrect, then `ServerVerify` messages generated with random keys. If correct, multiple handshakes use the same key.
- A protocol flaw!

Using an SSLv2 server as a Bleichenbacher oracle

- Attacker wishes to learn whether test ciphertext c has valid PKCS padding.
- Server sends `ServerVerify` message before client authenticates!
- If padding incorrect, then `ServerVerify` messages generated with random keys. If correct, multiple handshakes use the same key.
- A protocol flaw!

1. Attacker makes two SSLv2 connections with ciphertext c.
2. Attacker receives two `ServerVerify` messages.
3. For 40-bit export ciphers, can brute force symmetric key for one `ServerVerify`, check whether second uses same key.
4. Attacker has learned 2 most significant + 6 least significant bytes of plaintext:
   `m = 00 02 [padding] 00 [`mk_{secret}`]`

SSLv2 as a Bleichenbacher oracle

So SSLv2 is vulnerable to a Bleichenbacher oracle attack against an adversary who can brute force 2^{40} keys.

But this adversary could just brute force the key from any connection and skip RSA.

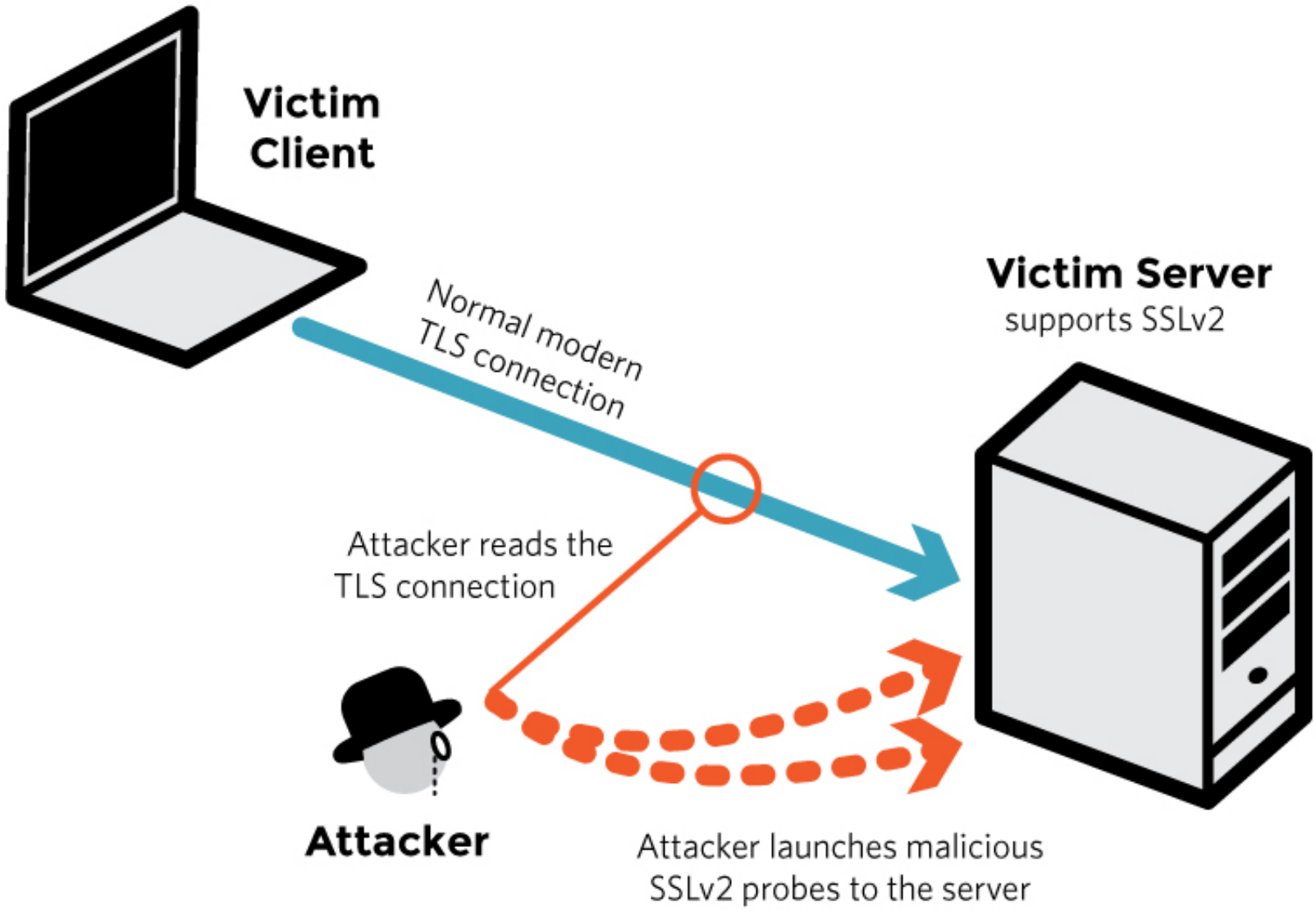
SSLv2 as a Bleichenbacher oracle

So SSLv2 is vulnerable to a Bleichenbacher oracle attack against an adversary who can brute force 2^{40} keys.

But this adversary could just brute force the key from any connection and skip RSA.

Observation: Servers use the same certificate/RSA public key for all SSL/TLS protocol versions.

DROWN: Using SSLv2 to decrypt TLS



DROWN attack: Use SSLv2 Bleichenbacher oracle to decrypt TLS

1. Attacker captures TLS ciphertexts from modern client-server connections that use RSA key exchange.

2. Attacker malleates TLS RSA ciphertext to SSLv2 export RSA ciphertext.

3. Attacker uses SSLv2 Bleichenbacher oracle to decrypt malleated ciphertext.

4. Attacker inverts ciphertext transformation to recover TLS secret keys, decrypts session.

Step 2: Transforming TLS ciphertexts to SSLv2 ciphertexts

- Challenge: TLS RSA ciphertexts have 48-byte message, SSLv2 export has 5-byte message.

Step 2: Transforming TLS ciphertexts to SSLv2 ciphertexts

- Challenge: TLS RSA ciphertexts have 48-byte message, SSLv2 export has 5-byte message.

- Solution: Given TLS ciphertext c, find s such that $s^e c$ is a valid SSLv2 ciphertext.

Step 2: Transforming TLS ciphertexts to SSLv2 ciphertexts

- Challenge: TLS RSA ciphertexts have 48-byte message, SSLv2 export has 5-byte message.
- Solution: Given TLS ciphertext c, find s such that $s^e c$ is a valid SSLv2 ciphertext.
- Challenge: For randomly chosen s,

$$\Pr[s^e c \text{ is SSLv2 conformant}] \approx 2^{-25}.$$

Step 2: Transforming TLS ciphertexts to SSLv2 ciphertexts

- Challenge: TLS RSA ciphertexts have 48-byte message, SSLv2 export has 5-byte message.

- Solution: Given TLS ciphertext c, find s such that $s^e c$ is a valid SSLv2 ciphertext.

- Challenge: For randomly chosen s,

$$\Pr[s^e c \text{ is SSLv2 conformant}] \approx 2^{-25}.$$

- Solution: Use *fractions*. ([Bardou et al.] "trimmers")

  Let $m = c^d \bmod N$. Assume m is divisible by some small t as an integer. If we try $s = u/t = ut^{-1} \bmod N$, then $sm \approx m$. (Thus most significant bits likely unchanged.)

- Requires $\approx 10,000$ oracle queries for 2048-bit RSA.

Step 3: Decrypting SSLv2 RSA ciphertext

- Challenge: Need to brute force 2^{40} RC2 keys.
- Solution: Naive CPU implementation: $\approx$ 10 core-hours. (In 1995, took 8 days on "120 workstations and a few parallel computers".)

Step 3: Decrypting SSLv2 RSA ciphertext

- Challenge: Need to brute force 2^{40} RC2 keys.
- Solution: Naive CPU implementation: $\approx$ 10 core-hours. (In 1995, took 8 days on "120 workstations and a few parallel computers".)
- Challenge: 100,000 core hours is a lot.
- Better solution: Optimized GPU brute forcer. 18 hours on 40-GPU cluster for full attack. ($\approx 2^{50}$ with optimizations.)
- Fun solution: On 350-GPU cluster on Amazon EC2, 8 hours and $440 for full attack.

Step 3: Decrypting SSLv2 RSA ciphertext

- Opportunity: Successful oracle query gives us 48 least significant bits of plaintext and 16 most significant bits of plaintext.
  $\mathtt{m}$ = 00 02 [padding] 00 [mk_{secret}]
  $c = m^e \bmod N$

- Solution: Multiply by $2^{-48} \bmod N$ to shift known plaintext bytes to most significant bytes.

$$
\begin{aligned}
m \cdot 2^{-48} \bmod N &= 2^{-48} \cdot (00\ 02\ [\text{padding}]\ 00\ [mk_{secret}]) \bmod N \\
&= 2^{-48} \cdot 00\ [mk_{secret}] \bmod N \leftarrow (\lg N\text{-bit known}) \\
&\quad + 00\ 02\ [\text{padding}] \leftarrow (\lg N - 64\text{-bit unknown})
\end{aligned}
$$

DROWN attack stages

1. Attacker collects many TLS ciphertexts.
2. Attacker applies fractions/trimmers to ciphertexts and queries SSLv2 server oracle until valid SSLv2 ciphertext found.
3. Attacker uses shifting trick + Bleichenbacher attack to decrypt SSLv2 ciphertext.
4. Attacker inverts fraction to recover decrypted TLS ciphertext and decrypts TLS session.

DROWN attack complexity for 2048-bit RSA

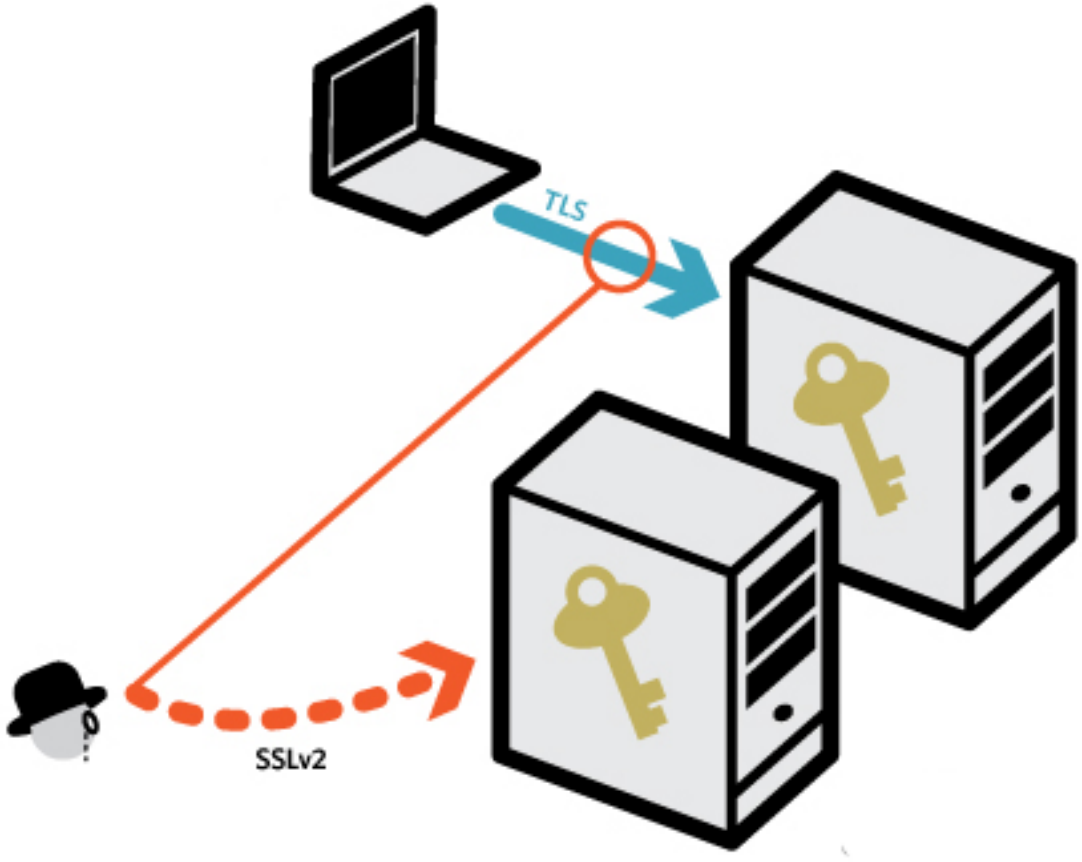
Optimizing	**Ciphertexts**	**Fractions**	**SSLv2 connections**	**Offline work**
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

How many servers were vulnerable to DROWN?

- At disclosure, 1.7M (10%) of HTTPS servers with browser-trusted certificates supported SSLv2.

How many servers were vulnerable to DROWN?

- At disclosure, 1.7M (10%) of HTTPS servers with browser-trusted certificates supported SSLv2.
- However, many more were vulnerable, due to key reuse across servers and *across protocols*.



How many servers were vulnerable to DROWN?

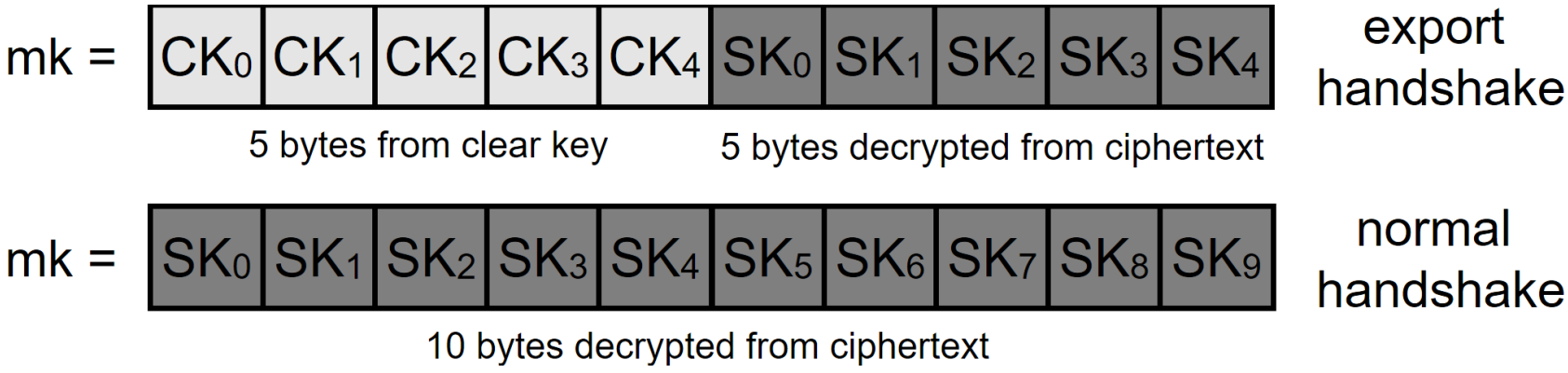
- At disclosure, 1.7M (10%) of HTTPS servers with browser-trusted certificates supported SSLv2.
- However, many more were vulnerable, due to key reuse across servers and *across protocols*.

Protocol		*All Certificates*			*Trusted certificates*		
		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 1M)		611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

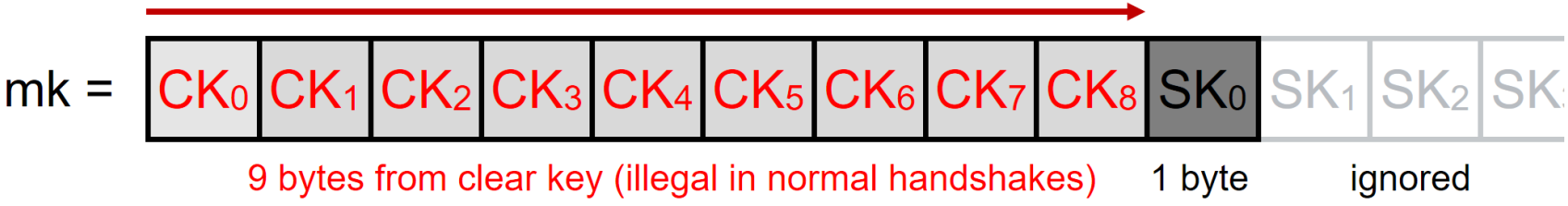
- Overall, 22% of HTTPS servers with trusted certs (25% of the Top Million) were vulnerable to DROWN.

CVE-2016-0703: OpenSSL "extra clear" vulnerability

Present in all OpenSSL versions from 1998 to March 2015



- Buggy OpenSSL would accept clear-key bytes in non-export handshakes, *displace* the secret key from the ciphertext.



- Unless ciphertext is not PKCS#1.5 compliant!
  Then *mk* is random.

An ideal oracle for DROWN

- Only requires one SSLv2 connection per oracle query, no large computation. (Attacker provides 10 bytes of *ck*, tests whether `ServerVerify` is encrypted with $mk = ck$.)
- When ciphertext is compliant, allows *sk* to be brute forced one byte at a time. (Attacker provides 9 bytes of *ck*, checks each of 256 possibilities for last byte of *mk*. Repeats with 8 bytes, etc.)

Special DROWN: Exploits OpenSSL vulnerability for even more devastating attack

- Script-kiddieable Bleichenbacher oracle!
  - 2^{50} computation per connection down to 2^{10} computation, no GPUs required.
  - Full attack can decrypt one in 260 TLS session keys using 17,000 server connections, < 1 minute, from a single workstation.

- Insight: This speed enables a MiTM attacker to attack TLS *onine*, before the handshake completes.
  - Even if server prefers PFS ciphers, can downgrade to RSA key exchange and obtain session key.
  - Can attack domain.com so long as any server with a valid cert for that name has the OpenSSL bug—doesn't have to use the same key.

- $\approx$ 66% of DROWN-vulnerable hosts vulnerable to this attack.

The DROWN Attack

To check whether your server appears to be vulnerable, enter the domain or IP address:

yahoo.com | Check for DROWN vulnerability

This tool shows vulnerable services detected in Feb. 2016 and probes each again to see if it's been fixed. Results won't include new servers or ones our scanner missed. Live updates are cached for 15 minutes.

Results for yahoo.com

The following domain names are vulnerable to **man-in-the-middle attacks**.
Attackers may be able to impersonate the server and steal or change data.

Update server software at all IP addresses shown, and ensure SSLv2 is disabled.

Vulnerable Domains:	Vulnerable Because:
accounts.yahoo.com edit.client.yahoo.com bt.edit.client.yahoo.com na.edit.client.yahoo.com verizon.edit.client.yahoo.com edit.yahoo.com *.edit.yahoo.com edit.europe.yahoo.com edit.india.yahoo.com edit.korea.yahoo.com login.korea.yahoo.com legalredirect.yahoo.com login.yahoo.com *.login.yahoo.com	206.190.42.37:443 vulnerable to CVE-2016-0703 currently vulnerable 98.137.205.232:443 ~~supports SSLv2 export ciphers~~ currently vulnerable to CVE-2016-0703

DROWN mitigations

- Update OpenSSL.
  OpenSSL team patched several bugs, disabled SSLv2 by default.
  One month after disclosure, only 15% of HTTPS hosts had patched!
- Fully disable SSLv2.
  Don't only disable export ciphers. If only ciphers are disabled, make sure they're actually disabled (CVE-2015-3197).
- Have single-use keys.
  Prudent to use different keys across different protocols and protocol versions.

Technical Lessons

- Obsolete cryptography considered harmful.

  Maintaining support for old services for backward compatibility isn't harmless.

- Limit complexity.

  Cryptographic APIs and state machines often overly complex.
  Design protocols to limit implementation mistakes.
  Design APIs to limit usage mistakes.

- Weakened cryptography considered harmful.

  Twenty years later, all three forms of SSL/TLS export crypto led to devastating attacks:

  - Export RSA (FREAK attack)
  - Export DHE (Logjam)
  - Export symmetric (DROWN).

Lessons for Policy

- *Technical backdoors in our infrastructure don't go away even when the political environment changes.*

  Twenty years of computing progress has brought attacks within range of modest attackers.

- Cannot assign cryptography based on nationality.

- Technological evidence opposes backdooring cryptography.

  Complexity of export cipher suites seems particularly prone to implementation vulnerabilities.

The good news: TLS can be secure

- TLS 1.2 with good choice of ciphers can be secure.[1]
- TLS 1.3 aggressively banning bad options.
  - Eliminating RSA key exchange.
  - Mminimum 2048 bits for FF-DHE.

[1] . . . as far as is publicly known, assuming implementations are correct, and assuming domain and key aren't exposed elsewhere in a weaker configuration.

A Messy State of the Union: Taming the Composite State Machines of TLS Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Jean Karim Zinzindohoue. *Oakland 2015*.

Factoring as a Service Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. *FC 2016*. `seclab.upenn.edu/projects/faas/`

Tracking the Freak Attack Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. `freakattack.com`

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. *CCS 2015*. `weakdh.org`

DROWN: Breaking TLS using SSLv2 Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. To appear in *Usenix Security 2016*. `drownattack.com`

The legacy of export-grade cryptography in the 21st century

Nadia Heninger and J. Alex Halderman

University of Pennsylvania University of Michigan

June 9, 2016