In the beginning, there was nothing.

1994

Netscape Navigator

Mosiac rebrands and releases Netscape Navigator.

1995

SSLv2

February, 1995. Netscape develops SSLv2. It is released, but found to be vulnerable to length extension and MITM attacks.

1995

Internet Explorer

Microsoft release IEv1 (rebranded, licensed Mosiac) in August, 1995.. IEv2 is released three months later.

1995

Apache Web Server

The open-source Apache httpd reaches 1.0 in December. It supports virtual hosting, and largely takes over the public web server market.

1996

SSLv3

Released by Netscape. Becomes the foundation for all of modern HTTPS. In 1999 the IETF adopts SSL and renames it to TLS.

1998

Microsoft IIS 4.0

First version of Microsoft's web server to support virtual hosting.

Secure Communication, Take 1

Confidentiality: Only the sender and receiver can read the message.

Integrity: The message has not been modified.

Authenticity: The message is from who you think it is.

What is it that you do here?

1. **History of SSL and TLS**
   What series of events led us to TLS 1.3? What motivated its design?

2. **What have we learned secure protocol design?**
   What have we learned in the last 30 years?

3. **Where is secure networking going?**
   TLS 1.3 was standardized. What comes next?

BUT WAIT…

UNIVERSITY OF MICHIGAN

Export Cryptography

In the 1990s, “export” of cryptography by US persons was regulated.*

International Traffic in Arms Regulations (ITAR), then Export Administration Regulations (EAR).

Ban on exporting code with a printed material exception.

Overturned during *Daniel J. Bernstein vs United States of America* in 1999 by the Clinton Administration.

*Technically it still is regulated, but most open-source software is exempt

Export Key Length Restrictions

Regulations applied to communication with non-US entities

Public-key Cryptography: Max 512-bit public keys

- Finite Field Diffie-Hellman (key exchange)
- RSA (key exchange, encryption)

Symmetric Cryptography: Max 40-bit keys

- Block ciphers (DES)
- Stream ciphers (RC4)

Signatures and Message Authentication Codes were *unregulated*

SSLv2

UNIVERSITY OF MICHIGAN

Client Hello: random, client-supported ciphers (…)

$\longrightarrow$

Client selects cipher

Server Hello: random, server ciphers (…), certificate

$\longleftarrow$

Client Master Key: cipher, mk_{clear}, $Enc_{PK}(\text{mk}_{secret})$

$\longrightarrow$

`write_key, read_key` = $\text{KDF}(\text{cipher}, mk_{clear} \,||\, mk_{secret})$

Server Verify: $Enc_{SWK}(random_{client})$

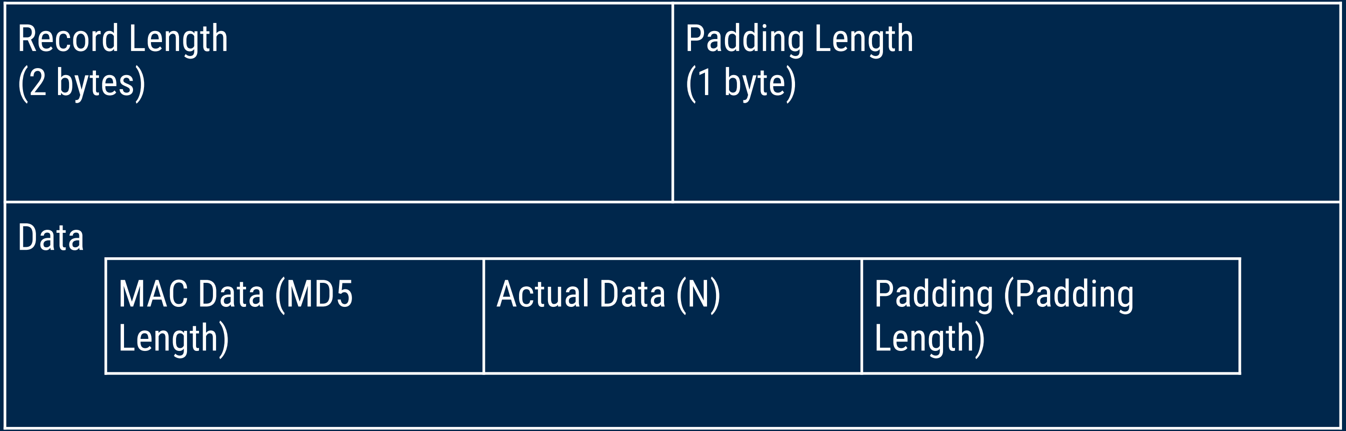
$\longleftarrow$

Client Finished: $Enc_{CWK}(random_{server})$

$\longrightarrow$

Server Finished: $Enc_{SWK}(session\_id)$

$\longleftarrow$

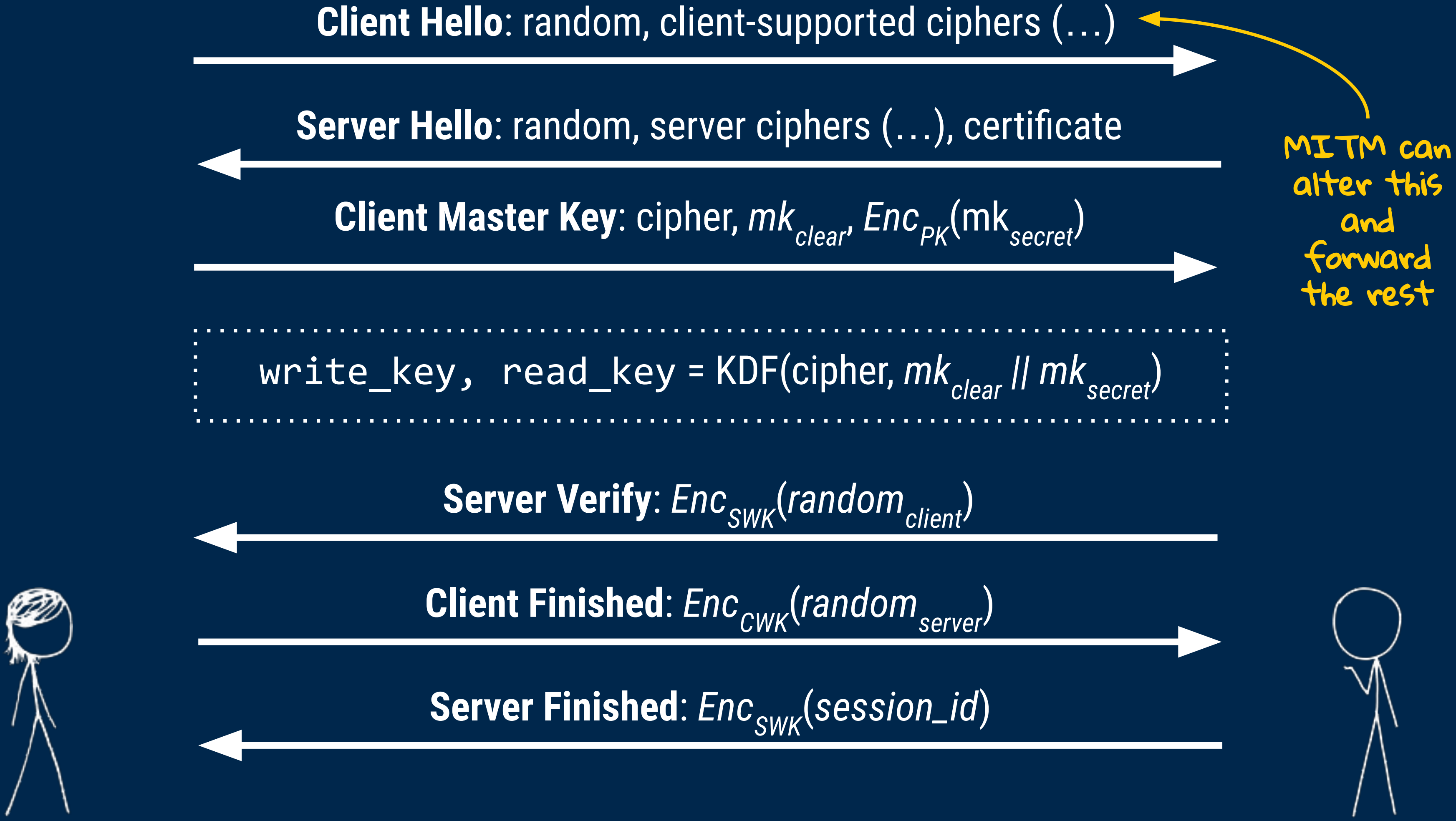


Record Length must be multiple of block size
Padding length is only if a block cipher is in use, pads to block length

MAC = MD5(secret, actual data, padding data, sequence number)
ENC-DATA = ENC(padding length, MAC, actual data, padding)

SSLv2 Problems

- No commitment to the handshake messages
  - MITM can force a downgrade without knowing the keys, including downgrade to export-grade ciphers
- Fixed to non-HMAC MD5 hash function
  - No collision resistance, does have preimage and second-preimage resistance
  - MAC is not an HMAC, it's just a keyed hash, so it's vulnerable to length extension
  - $HMAC(H, k, m) = H(k \| H(k \| m))$
- No concept of certificate chains, only leaf certificates
  - Could be a positive or a negative
- Only stream cipher is RC4
  - Known issues lowering security level below targets
- Block ciphers are all used in CBC mode
  - Padding oracles



Client Hello: random, client-supported ciphers (…)

Server Hello: random, server ciphers (…), certificate

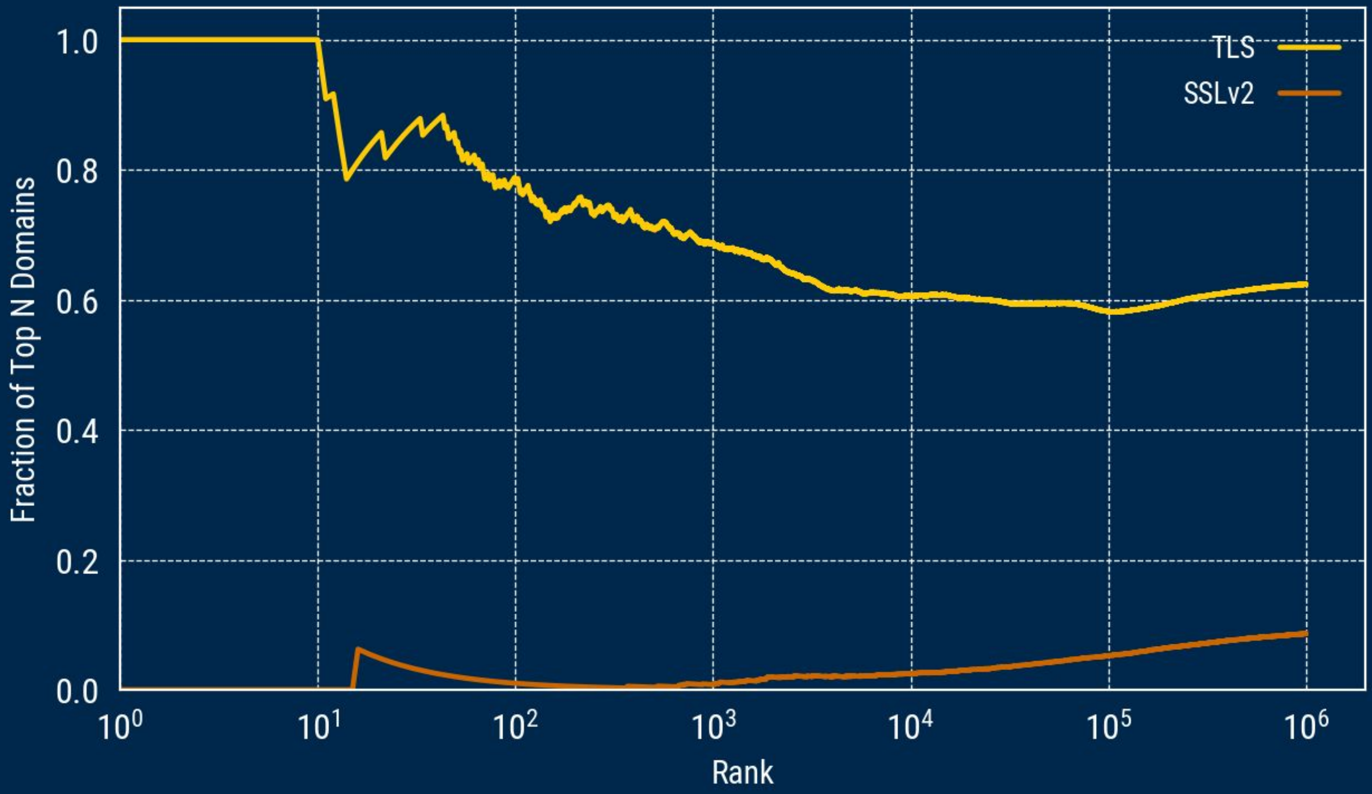
Client Master Key: cipher, mk_{clear}, $Enc_{PK}(\mathrm{mk}_{secret})$

`write_key, read_key` = KDF(cipher, $mk_{clear} \mathbin{||} mk_{secret}$)

Server Verify: $Enc_{SWK}(random_{client})$

Client Finished: $Enc_{CWK}(random_{server})$

Server Finished: $Enc_{SWK}(session\_id)$



SSLv2 / TLS Support Among Top 1M Domains

(2016, pre-Drown Attack)

UNIVERSITY OF MICHIGAN

		All Certificates		Trusted Certificates	
Protocol	**Port**	**TLS**	**SSLv2**	**TLS**	**SSLv2**
SMTP	25	**3,357 K**	**936 K (28%)**	**1,083 K**	**190 K (18%)**
POP3	110	4,193 K	404 K (10%)	1,787 K	230 K (13%)
IMAP	143	4,202 K	473 K (11%)	1,781 K	223 K (13%)
HTTPS	443	34,727 K	5,975 K (17%)	17,490 K	1,749 K (10%)
SMTPS	465	3,596 K	291 K (8%)	1,641 K	40 K (2%)
SMTP	587	3,507 K	423 K (12%)	1,657 K	133 K (8%)
IMAPS	993	4,315 K	853 K (20%)	1,909 K	260 K (14%)
POP3S	995	4,322 K	884 K (20%)	1,974 K	304 K (15%)

SSLv2 Support in Non-HTTPS Protocols
(2016, pre-Drown Attack)

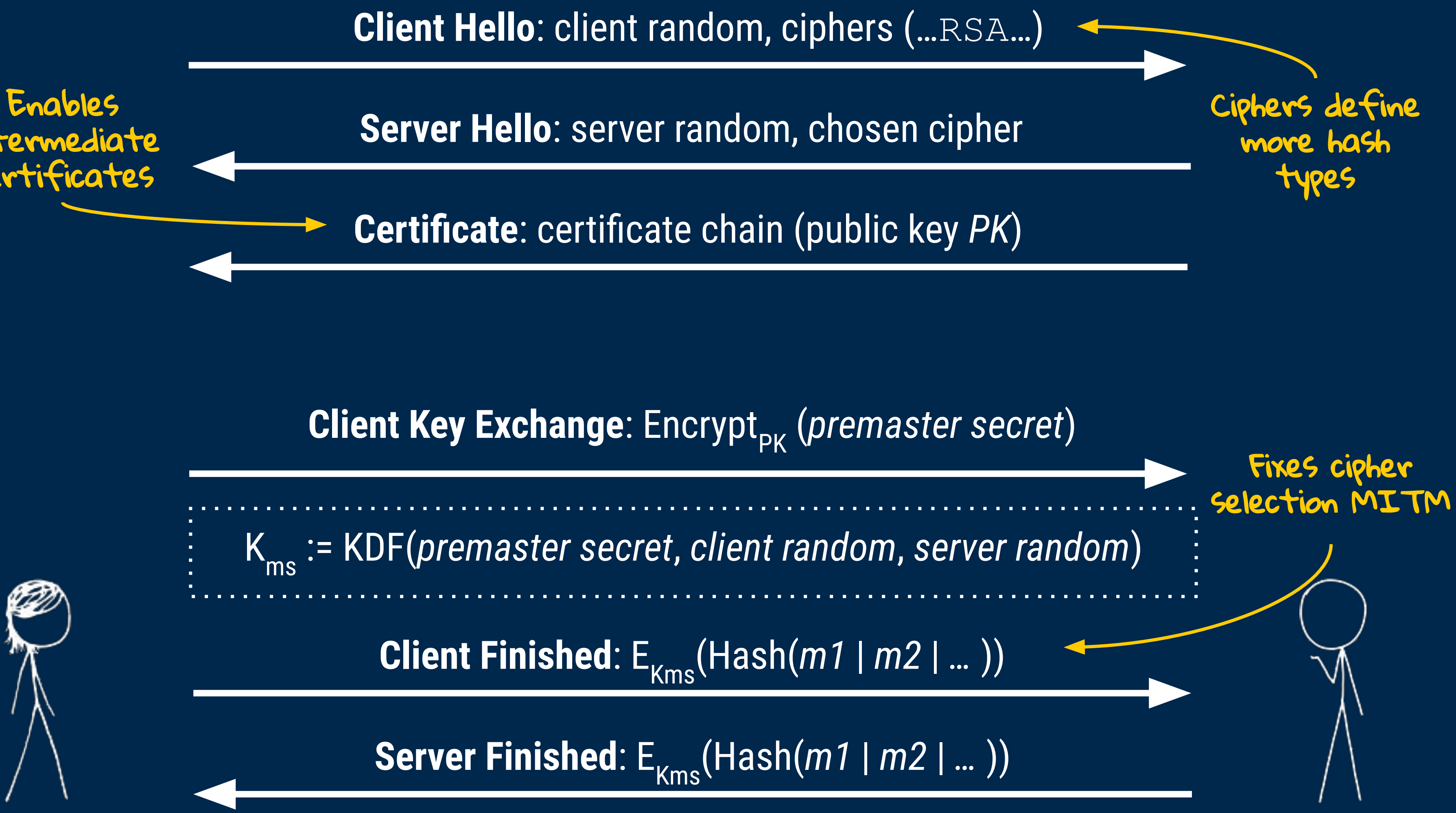
UNIVERSITY OF MICHIGAN

SSLv2 Good Stuff?

- Uses Key Encapsulation / Data Encapsulation (KEM/DEM)*
  - Use public keys to agree on a random number in secret (encrypt it)
  - Use random number to seed a KDF
  - Use KDF to derive a symmetric key
- Uses record layer with plaintext lengths*
  - Easy to figure out how big your buffer should be when implementing
- Doesn’t try to solve key distribution (leaves it for the certificate authorities and the browser)*

*exceptions exist

TLS



Client Hello: client random, ciphers (…RSA…)

Server Hello: server random, chosen cipher

Certificate: certificate chain (public key *PK*)

Client Key Exchange: Encrypt_{PK} (*premaster secret*)

K_{ms} := KDF(*premaster secret*, *client random*, *server random*)

Client Finished: $E_{Kms}(\text{Hash}(m1 \mid m2 \mid \dots))$

Server Finished: $E_{Kms}(\text{Hash}(m1 \mid m2 \mid \dots))$

Enables intermediate certificates

Ciphers define more hash types

Fixes cipher selection MITM

Extensibility and Agility

A protocol is **extensible** if you can add a feature to it without having to update every implementation at the same time.

- *Extensions* and *Version Number* provide extensibility for TLS

A protocol has **cryptographic agility** if implementations can negotiate the underlying cryptography in use.

- Cipher suites provide agility in TLS

Multiplexing on Names

If you have more than one service per host, you need to multiplexed by some identifier (usually name).

HTTP virtual hosting is powered by the `Host` header. TLS exposes this via the SNI extension (cleartext).

Any secure protocol has to answer:

- How does it multiplex?
- Is the identifier private or public?

Client Hello

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

[RFC 5246, TLS 1.2, Rescola]

Exercise for the reader: *Where are the incompatibility bugs?*

“HAVE ONE JOINT AND KEEP IT WELL OILED.”

ADAM LANGLEY, GOOGLE

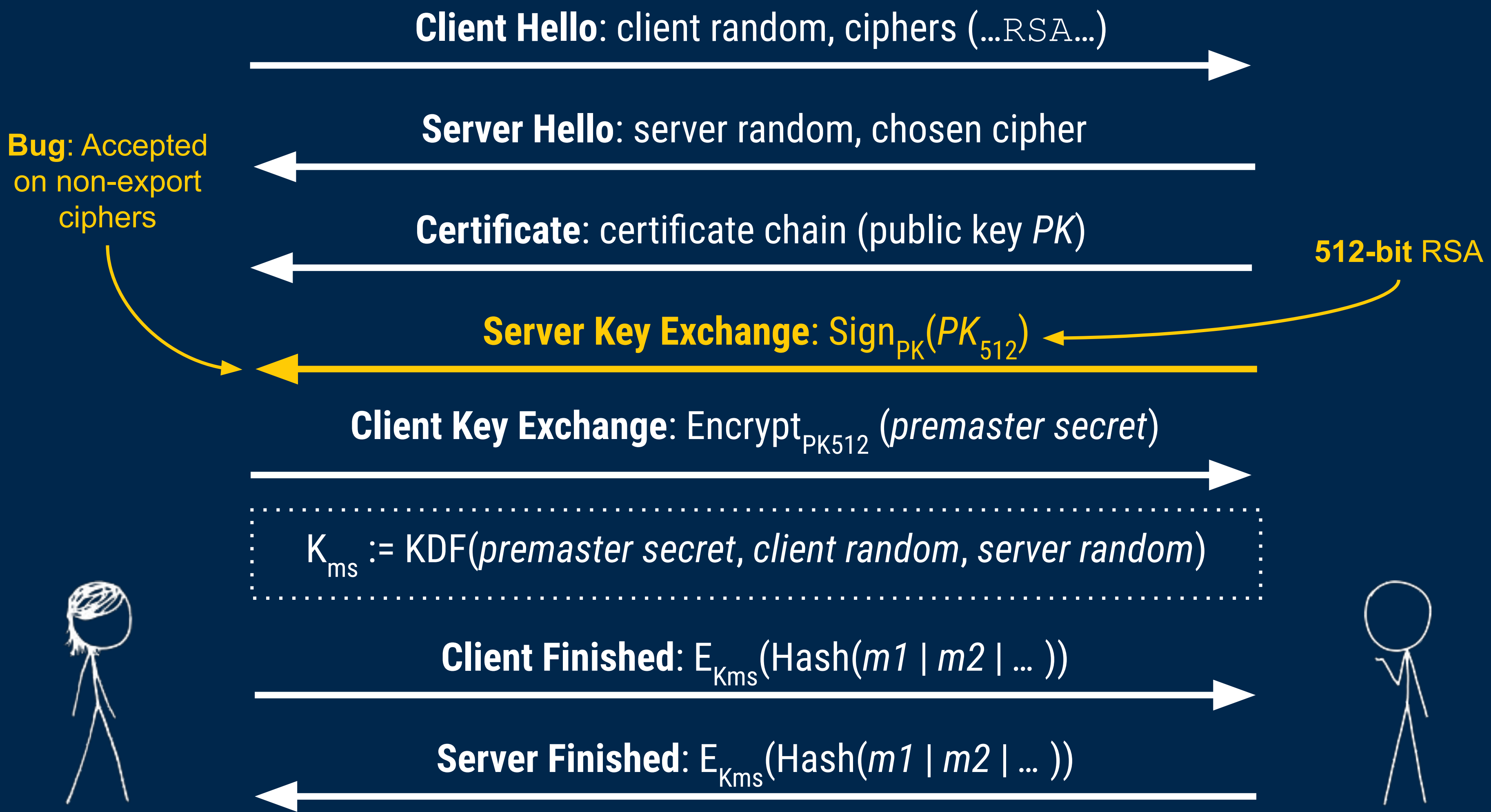
Cipher Suites

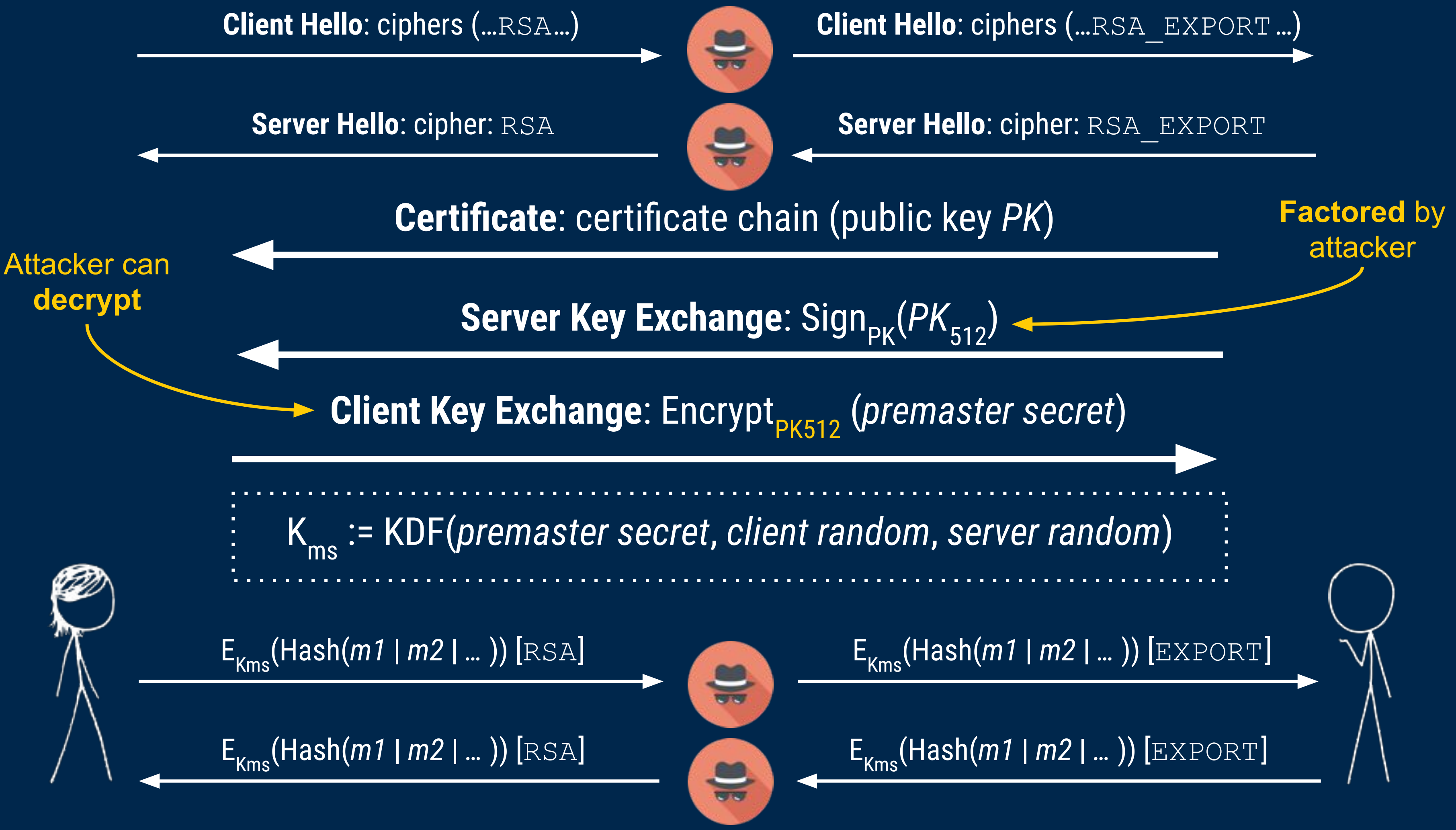
Define the key exchange, signature and hash (if needed), and symmetric encryption used for a connection.

```
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

TLS certificates contain >512-bit RSA keys

- OK for authentication!
- *Literally Illegal* for key exchange in the 1990s!


Client Hello: client random, ciphers (…RSA…)
Server Hello: server random, chosen cipher
Bug: Accepted on non-export ciphers
Certificate: certificate chain (public key *PK*)
512-bit RSA
Server Key Exchange: $\mathrm{Sign}_{PK}(PK_{512})$
Client Key Exchange: $\mathrm{Encrypt}_{PK512}$ (*premaster secret*)
K_{ms} := KDF(*premaster secret*, *client random*, *server random*)
Client Finished: $E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$
Server Finished: $E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$


Client Hello: ciphers (…RSA…)
Client Hello: ciphers (…RSA_EXPORT…)
Server Hello: cipher: RSA
Server Hello: cipher: RSA_EXPORT
Certificate: certificate chain (public key *PK*)
Factored by attacker
Attacker can **decrypt**
Server Key Exchange: $\mathrm{Sign}_{PK}(PK_{512})$
Client Key Exchange: $\mathrm{Encrypt}_{PK512}$ (*premaster secret*)
K_{ms} := KDF(*premaster secret, client random, server random*)
$E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$ [RSA]
$E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$ [EXPORT]
$E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$ [RSA]
$E_{Kms}(\mathrm{Hash}(m1 \mid m2 \mid \dots))$ [EXPORT]

FREAK Attack

Discovered by researchers at INRIA attempting to formally model TLS implementation state machines.

FREAK attack allows an attacker who can factor 512-bit RSA keys to man-in-the-middle TLS connections to servers that support export-grade RSA.

Computers are faster now than in 1998, 512-bit keys can be factored in **1-2 hours for ~$100** [Valenta 2015]

FREAK Impact

FREAK is an **implementation bug** stemming from complexity around compliance mechanisms for export regulations

- OpenSSL (Chrome)
- Microsoft SChannel (Internet Explorer)
- Apple SecureTransport (Safari)

Modern clients are vulnerable, but FREAK *only has impact if* servers support export-grade RSA ciphers.

Measure support for export-grade RSA!

FREAK Measurements

The FREAK attack is possible when a vulnerable browser connects to a susceptible web server—a server that accepts “export-grade” encryption.

	Vulnerable at Disclosure (March 3, 2015)	**Vulnerable One Week Later (March 10, 2015)**
HTTPS Servers at Alexa Top 1M Domains	9.6%	8.5%
HTTPS servers with browser-trusted certificates	36.7%	6.5%
All HTTPS servers	26.3%	11.8%

UNIVERSITY OF MICHIGAN

RSA Key Exchange

FREAK attack is not the only attack on RSA.

- Textbook RSA is broken
  - No semantic security (deterministic, 1:1 mapping from plaintext to ciphertext)
  - Can't encrypt anything larger or shorter than $n = pq$
  - Requires *padding* to implement securely
  - PKCS#1.5 padding is difficult to securely parse (timing attacks, padding oracle issues)
  - OAEP (encryption) and PSS (signatures) are easier to parse, may still have timing issues
- Bleichenbacher Padding Oracle
  - Extended to use SSLv2 servers to attack TLSv1.2 connections in the DROWN attack [Aviram 2016]
  - Attacks PKCS #1.5 padding
- Bleichenbacher *e=3* attack
  - A small plaintext might not wrap N, calculate the cube root to decrypt

Forward Secrecy

Forward secrecy is the property that compromising key material of a session does not allow you to decrypt past sessions.

Threat model is someone who is recording traffic or who has consistent access to your network traffic.

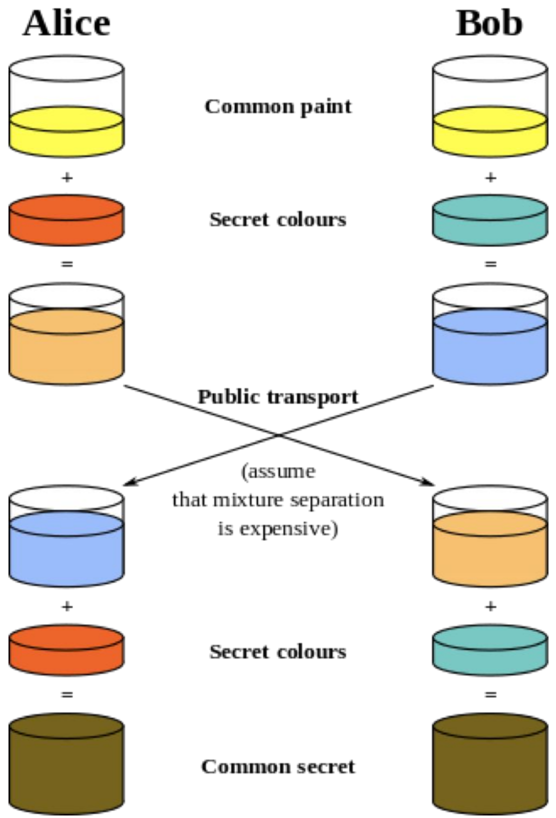
Forward secrecy means they would have to break each connection individually, or actively man-in-the-middle a connection, even if long-term key material is compromised.

Ephemeral Diffie-Hellman

TLS achieves forward-secrecy via **ephemeral** Diffie-Hellman.

Generating a new Diffie-Hellman key pair each connection to derive a master secret decouples the long-term key from the session key, providing forward secrecy.

The long-term key is only used for authentication. If an attacker compromises the long-term key, they cannot decrypt past sessions (forward secrecy).



Finite-Field Diffie-Hellman (FFDH)

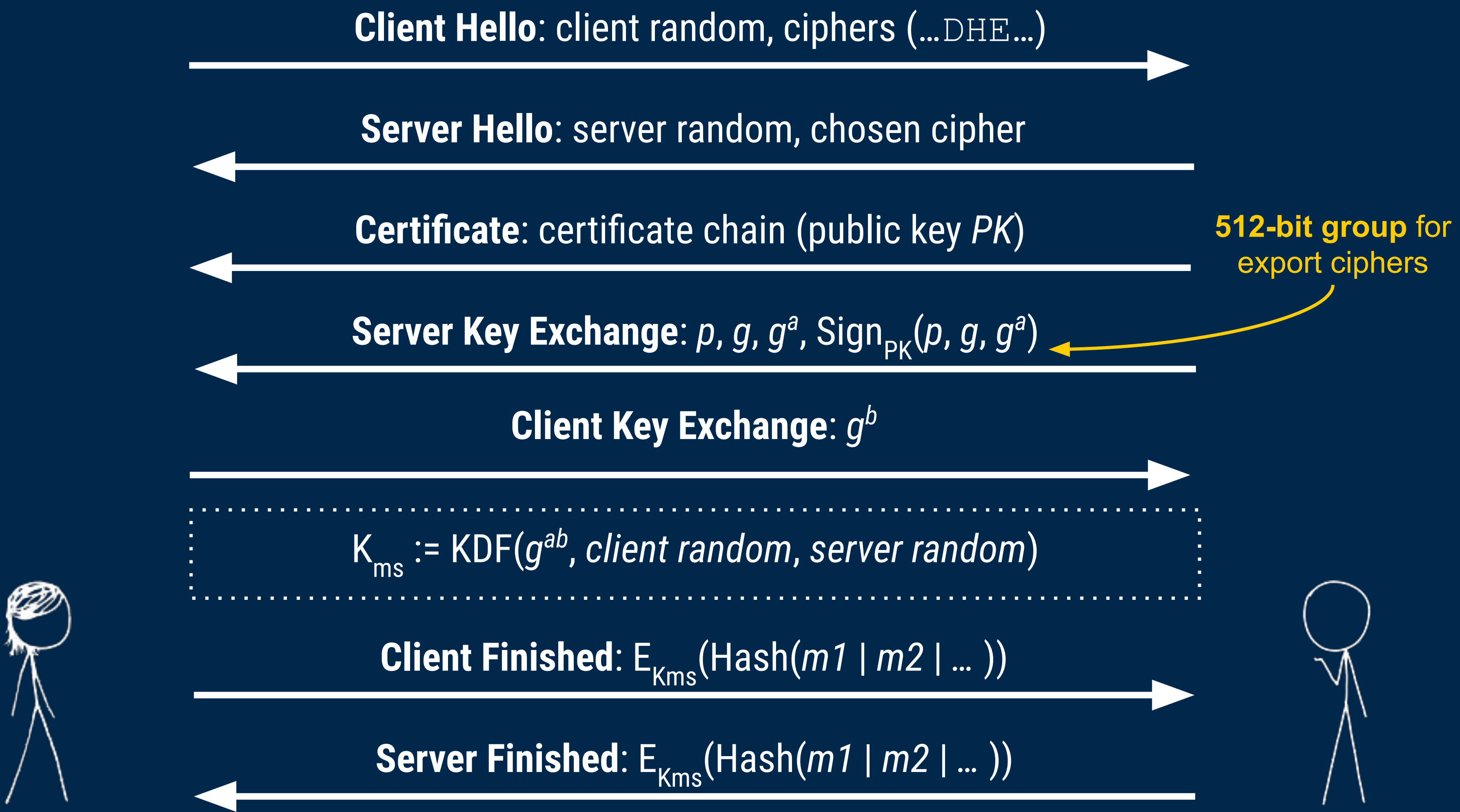
Let *(p, g, +, *)* define a field generated by *g* modulo *p,* where *p* is prime. Alice generates $x = g^a$, Bob generates $y = g^b$.

Alice sends *x*, and calculates y^a. Bob sends *y*, and calculates x^b. Then both sides calculate the secret $y^a = g^{ba} = g^{ab} = x^b$ mod p, assuming that log mod p is hard (discrete log).

Even though we have a field (two operations), we only need one operation. If we could find a group instead of a field, with similar properties, then we would have less algebraic structure and could get equivalent security at lower bit secrets (less computation).

Answer: *elliptic curves* (later)

See October 26, 2021 episode of “Security, Cryptography, Whatever”



Client Hello: client random, ciphers (…`DHE`…) →

Server Hello: server random, chosen cipher ←

Certificate: certificate chain (public key *PK*) ←

Server Key Exchange: p, g, g^a, $\text{Sign}_{PK}(p, g, g^a)$ ←

Client Key Exchange: g^b →

$K_{ms} := \text{KDF}(g^{ab}, \textit{client random}, \textit{server random})$

Client Finished: $E_{K_{ms}}(\text{Hash}(m1 \mid m2 \mid \ldots))$ →

Server Finished: $E_{K_{ms}}(\text{Hash}(m1 \mid m2 \mid \ldots))$ ←

Elliptic Curve Diffie-Hellman (ECDH)

Select $(C, g, +)$ where C is an elliptic curve, and g generates a group of points on the curve.

Instead of defining addition, multiplication and exponentiation, define addition and multiplication by a scalar as repeated addition ($3x = x + x + x$)

Alice and Bob generate secret scalars a and b. Alice publishes $x = ag$. Bob publishes $y = bg$.

Diffie-Hellman is now: $(a)y = (ab)g = (b)x = (ba)g$

The addition operation is a fancy formula that depends on the curve C.

Curves are defined in advance (e.g. X25519). Selecting curves with efficient and misuse-resistant formulas is an active area of research

ECDH in TLS 1.2

- Negotiated via cipher suite
- Supported curves are indicated in an extension
- Content of ClientKeyExchange and ServerKeyExchange is switched from FFDH to ECDH

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
```

Safely Using FFDH

Use elliptic curve Diffie-Hellman instead! (ECDH / ECDHE)

If that's not possible, see the last paragraph of [Valenta 2017]:

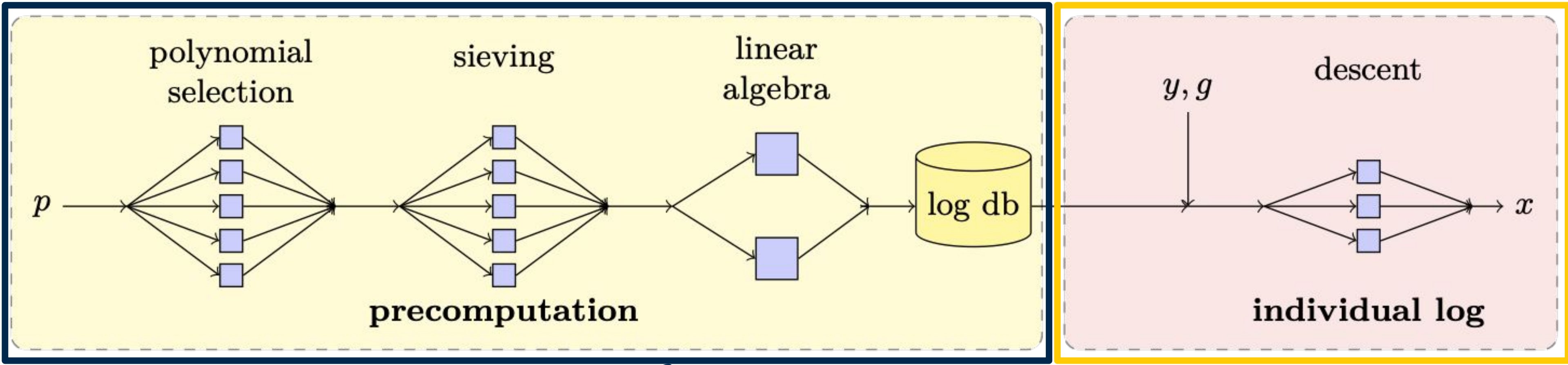
- Prefer safe primes, those where (p-1)/2 is also prime
- Use primes with documented provenance (not a trapdoor prime) [Fried 2016]
    - Don't let endpoints pick their own prime
- Validate key exchange value is between 1 and p-1, exclusive
- Use **ephemeral** key exchange values for every connection (don't reuse them)
- Use exponents of at least 224 bits (*a* and *b*)

See October 26th, 2021 episode of "Security, Cryptography, Whatever" podcast

Discrete Log

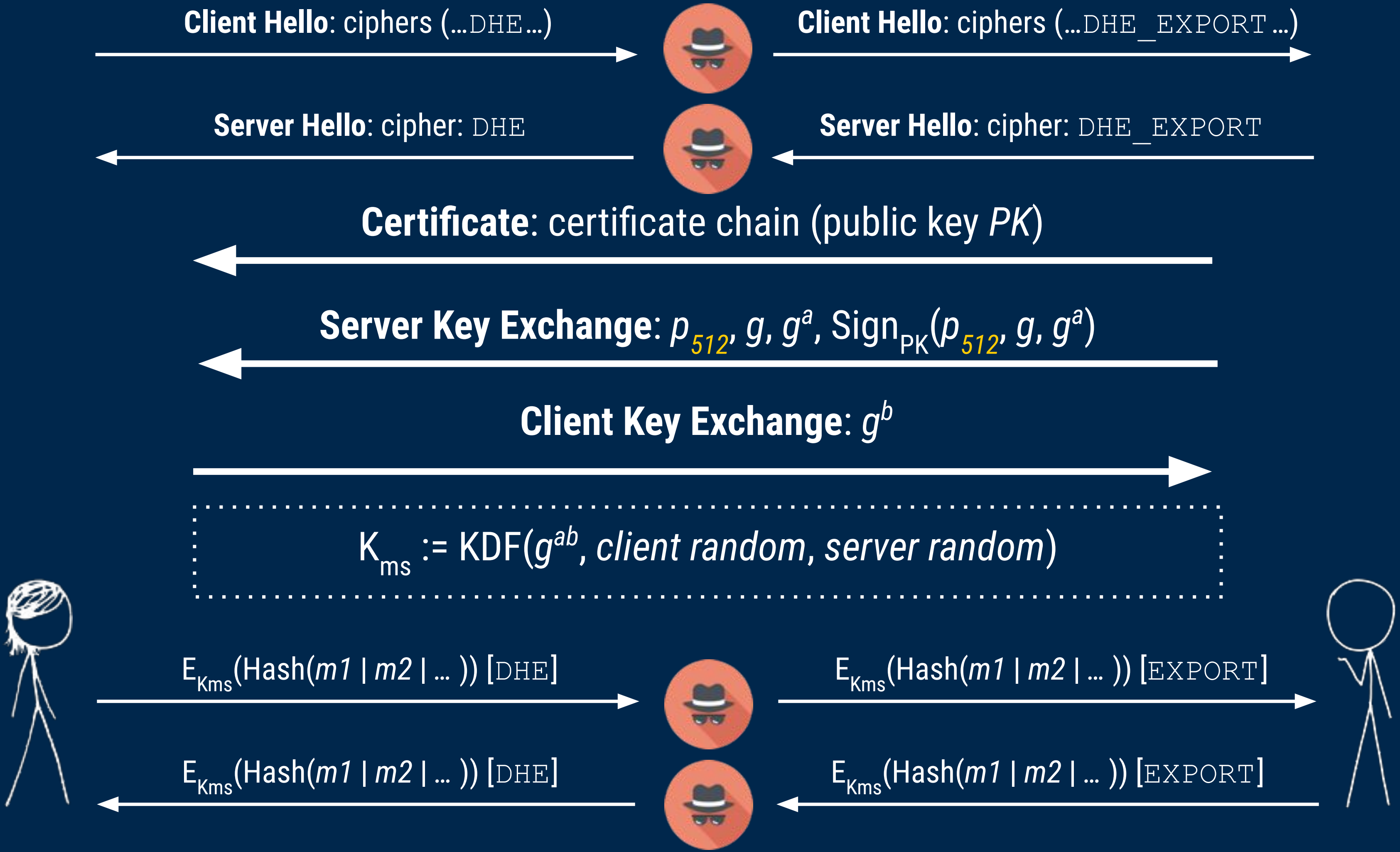
The security of Diffie-Hellman relies on the computational hardness of computing **discrete logs**, e.g. given *g*, *p*, and g^x mod *p*, calculate *x*.

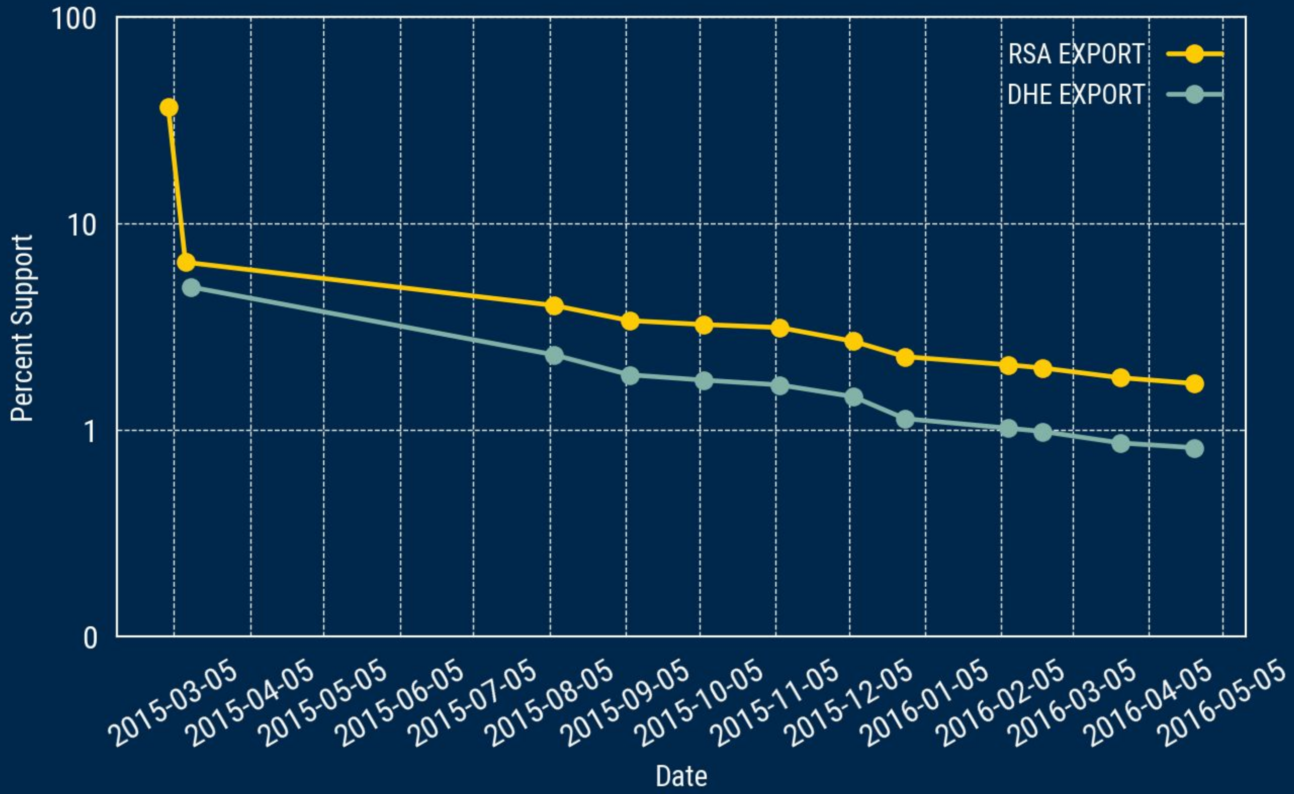
The **number-field sieve** is the fastest-known algorithm for computing discrete logs.



Depends only on p, 1 week for 512-bit p

Fast


Client Hello: ciphers (…DHE…)
Client Hello: ciphers (…DHE_EXPORT…)
Server Hello: cipher: DHE
Server Hello: cipher: DHE_EXPORT
Certificate: certificate chain (public key *PK*)
Server Key Exchange: p_{512}, g, g^a, $\text{Sign}_{PK}(p_{512}, g, g^a)$
Client Key Exchange: g^b
K_{ms} := KDF(g^{ab}, *client random, server random*)
E_{Kms}(Hash(*m1* | *m2* | …)) [DHE]
E_{Kms}(Hash(*m1* | *m2* | …)) [EXPORT]
E_{Kms}(Hash(*m1* | *m2* | …)) [DHE]
E_{Kms}(Hash(*m1* | *m2* | …)) [EXPORT]



Export Cipher Support Among Servers with Trusted Certificates

UNIVERSITY OF MICHIGAN

Top 1M Support

8.5% of the Alexa Top 1M supported DHE_EXPORT
3.4% of trusted IPv4 supported DHE_EXPORT

Prime	**Popularity Among Top 1M**
Apache mod_ssl prime	82%
Nginx prime	10%
Other (463 primes)	8%

Cipher Suite Agility Doesn’t Save You From Backdoors

FREAK is an implementation vulnerability surrounding export-grade cryptography,

Logjam is a protocol vulnerability where the difficulty to exploit relies on empirical properties of the Internet.

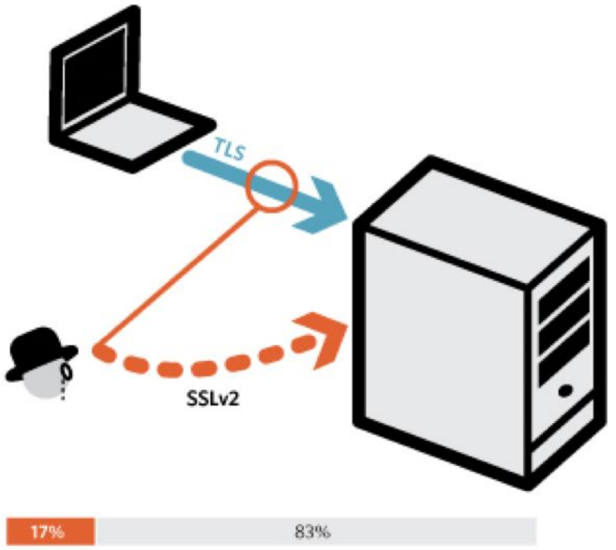
In both cases, (mis?)behavior of hosts on the Internet contributed to the vulnerability (sharing primes, reusing 512-bit keys, consistent cross-implementation bugs)

DROWN also exists and uses export ciphers in SSLv2 to attack TLS 1.2 [Aviram 2016]

DROWN Vulnerability

A server is vulnerable to DROWN if:

It allows both TLS and SSLv2 connections



17% of HTTPS servers still allow SSLv2 connections

Mitigations and Lessons from DROWN

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

- Usually discussed in the context of signatures vs. encryption
- Prudent to use different keys across different protocol versions
- Domain separation helps

Authenticate the client before sending secret-derived data

- DROWN is possible because of the early `ServerVerify` message
- Design protocols to check the client has knowledge of the secret first

All types of export cryptography have led to attacks against modern cryptography.

UNIVERSITY OF MICHIGAN

TLS 1.3

Authenticated Encryption with Associated Data (AEAD)

AEAD ciphers exist for TLS 1.2 and earlier, however all TLS 1.3 ciphers MUST be AEADs. An AEAD has the following API:

`Seal(key, nonce, associated_data, plaintext)` `-> (ciphertext)`, where `len(ciphertext) = len(plaintext) + TAG_LEN`, and nonce has fixed `NONCE_LEN`.

`Open(key, nonce, associate_data, ciphertext)` `-> (plaintext)`, where `len(plaintext) = len(ciphertext) - TAG_LEN`, and nonce has fixed `NONCE_LEN`. If authentication fails, than an error is returned.

Example AEADs: AES-GCM, ChaCha-Poly1305, Kravatte, Xoodyak,

AEAD Usage

Why are AEADs nice for secure transports?

- Use a plaintext packet header as associated data. This header can contain length, message type, etc.
- An AEAD functions like a stream cipher. No padding is needed. The output length always matches the input length, plus the tag length (MAC).
- Nonces can be sent in plaintext, and randomly generated. Don’t reuse them

Issues with AEADs

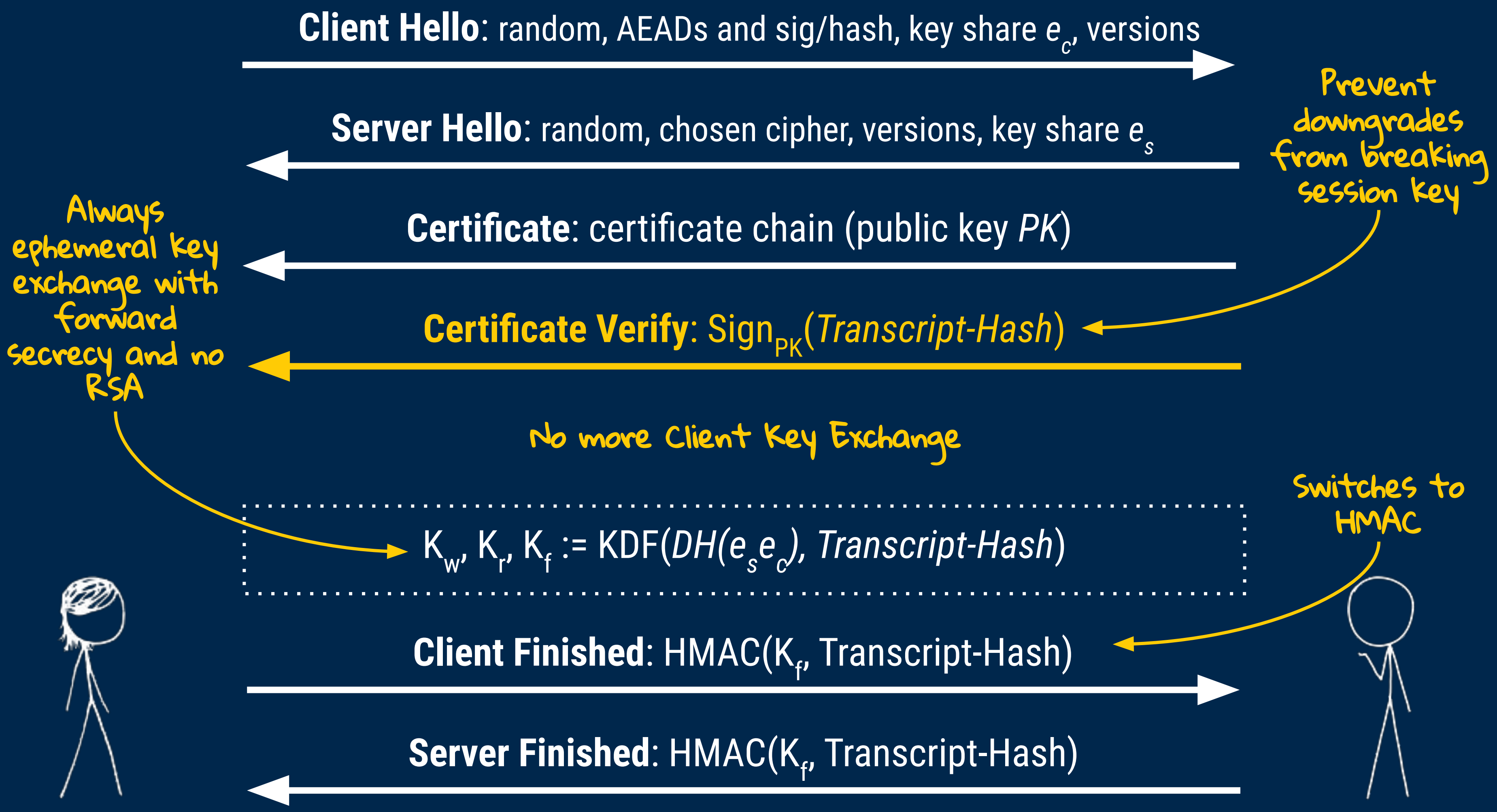
- Nonce management is dependent on the AEAD (not all need them). Some have catastrophic failures if a nonce is reused.

TLS 1.3

Goals:
1. Reduce handshake latency
2. Encrypt more of the handshake
3. Improve resilience to cross-protocol and downgrade attacks
4. Remove legacy features

TLS 1.3 was standardized in August, 2018. It began development in August, 2013.
Informed by the last 10 years of cryptography research. Can it avoid the pitfalls of the past?

New “version”, but with a very new shape. [RFC 8446, Rescola, 2018]



Client Hello: random, AEADs and sig/hash, key share e_c, versions

Server Hello: random, chosen cipher, versions, key share e_s

Certificate: certificate chain (public key *PK*)

Certificate Verify: $\text{Sign}_{PK}(\textit{Transcript-Hash})$

No more Client Key Exchange

$K_w, K_r, K_f := \text{KDF}(\textit{DH}(e_s e_c), \textit{Transcript-Hash})$

Client Finished: $\text{HMAC}(K_f, \text{Transcript-Hash})$

Server Finished: $\text{HMAC}(K_f, \text{Transcript-Hash})$

Prevent downgrades from breaking session key

Always ephemeral key exchange with forward secrecy and no RSA

Switches to HMAC

TLS 1.3 Key Exchange

Named Groups

- Unify parameter selection for (EC)DHE
- Select from preset list of curves/groups, each following best practices
- **1-RTT Mode**: Client guesses supported group and provides key share during Hello. Server responds with its contribution, or a HelloRetryRequest and a list of groups.

Limiting RSA

- No RSA key exchange
- Replace PKCS#1.5 with PSS to avoid Bleichenbacher failures

Explicit Verification

- Server signs handshake transcript with certificate key
- Prevents downgrade attacks leveraging weak session keys

TLS 1.3 Client Hello

Problem: Needs to look like a TLS 1.2 Client Hello for compatibility reasons, but work in new ways with 1.3 servers.

Solution: TLS 1.3 only cipher suites, move version negotiation and key share to an extension, deprecate old fields. The protocol can diverge from old versions after the Client Hello.

```
struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```

TLS 1.3 Server Hello

Problem: Needs to look like a TLS 1.2 Server Hello for compatibility reasons with middleboxes, but work in new ways with 1.3 clients.

Solution: TLS 1.3 only cipher suites, move version negotiation and key share to an extension, deprecate old fields. Sentinel random indicates `HelloRetryRequest` when necessary. Fix the last eight bytes of the randomness when negotiating TLS <=1.2 to prevent downgrades.

```
struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
  Random random;
  opaque legacy_session_id_echo<0..32>;
  CipherSuite cipher_suite;
  uint8 legacy_compression_method = 0;
  Extension extensions<6..2^16-1>;
} ServerHello;
```

TLS 1.3 0-RTT Mode

By agreeing on a PSK to use with future connections, it is possible for a client to being future connections before waiting for a server response [RFC 8446, Rescola, 2018]

These messages are replayable. This could lead to security flaws. The spec says not to handle requests that modify data until after the replay window is up (after the server finishes the handshake).

Functionality primarily used by “Big Tech”

This is the sketchiest part of all of TLS 1.3. Someone should measure this. Maybe ICSI has?

NOT TLS

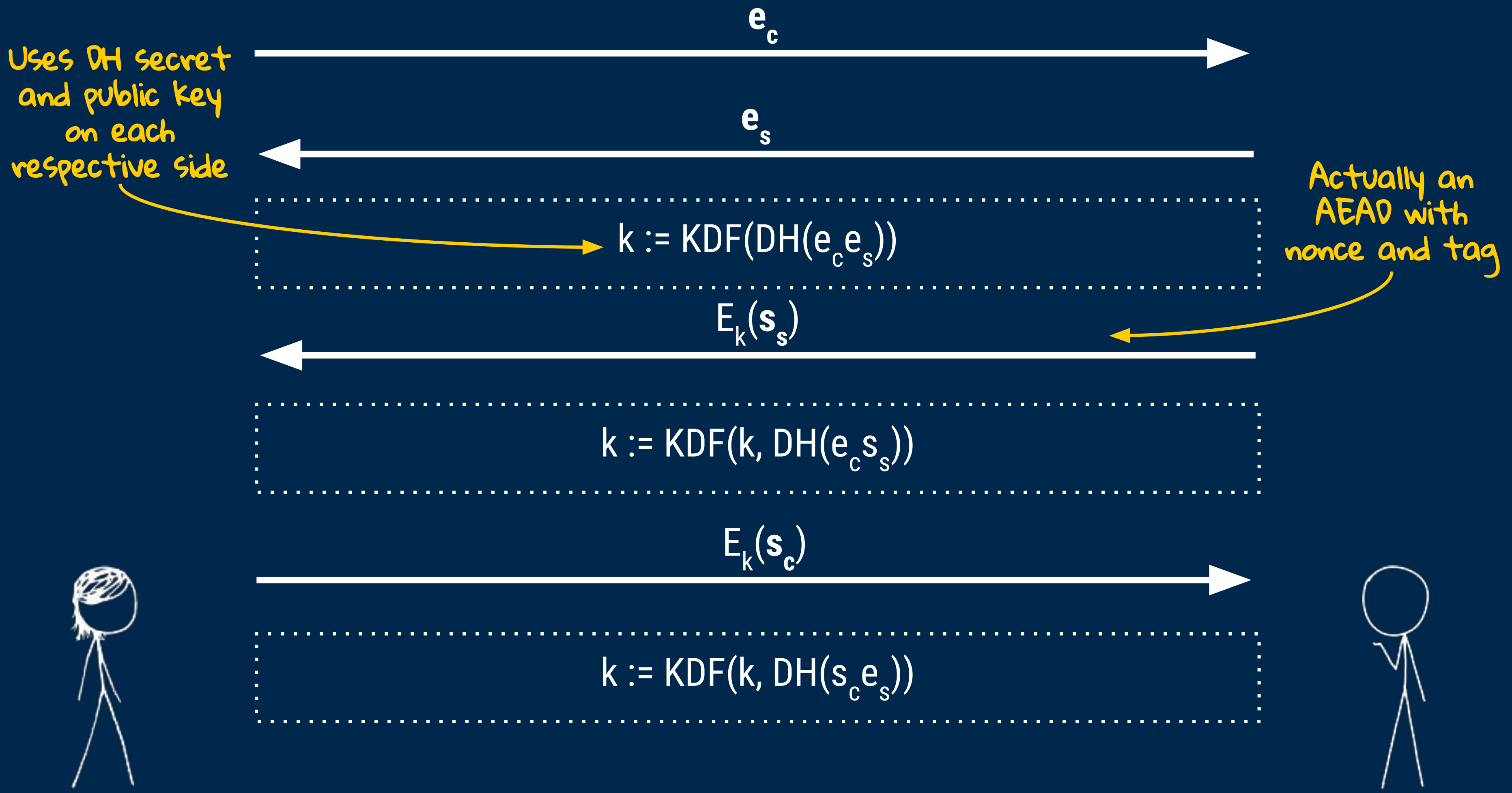
UNIVERSITY OF MICHIGAN

Noise Protocol Framework

Noise is a set of guidelines for describing protocols for authenticated secure channels using Diffie-Hellman as the only asymmetric primitive, combined with an AEAD.

There are no signatures!

The two parties are an **initiator** and a **responder.**


$\mathbf{e_c}$
$\mathbf{e_s}$
$k := KDF(DH(e_c e_s))$
$E_k(\mathbf{s_s})$
$k := KDF(k, DH(e_c s_s))$
$E_k(\mathbf{s_c})$
$k := KDF(k, DH(s_c e_s))$
Uses DH secret and public key on each respective side
Actually an AEAD with nonce and tag

Noise Notation

XX handshake pattern (mutually authenticated by a long-term key)

```
-> e
<- e, ee, s, es
-> s, se
```

NN handshake pattern (anonymous)

```
-> e
<- e, ee
```

Why Triple DH?

-> s

<- s, ss

No **forward secrecy** in static-static.

-> e

<- s, es

-> s, se

Avoid **Key Compromise Impersonation (KCI)** attacks against a compromised long-term key, where the attacker impersonates another party to the victim with the compromised key. Any side that contributes a static key must also contribute an ephemeral.
https://romailler.ch/2021/08/18/crypto-why-ephemeral-keys/

WHAT DID WE LEARN?

UNIVERSITY OF MICHIGAN

A Look Back on SSLv2: The Good Parts

TLS 1.3 fully drops the RSA KEM/DEM design inherited from SSLv2.
Not a knock on all KEM/DEM, but we have better ways of doing key agreement (DH).

Switching to AEADs makes it even easier to have authenticated plaintext data associated with a encrypted payload
Plaintext header data continues to make protocol implementation easier

For better or for worse, we still use X.509 certificates as the primary Web PKI.
Stare not into the abyss, lest you become recognized as an abyss domain expert, and they keep expecting you to stare into the damn thing.

How to Design and Build A Secure Transport

Who is using the protocol and **why** are they using it?

1. Can you just use TLS?
2. If not, can you use a specific Noise instantiation?

More Questions

If the answer is “no” to both of those:

- TCP vs UDP? Is there IP roaming? Connection-resumption? Long-lived connections?
- Do you control all the endpoints? Can you do full-fleet upgrades? Can you run two versions simultaneously? Can you run two servers simultaneously? Can you have more than one endpoint?
- What hardware constraints do you have? Can you use OpenSSL? Libcrypto? Libsodium? Are there binary size constraints? What are the memory constraints?
- What cryptography can be hardware accelerated? What has to be done in software?
- What is the key distribution method? Do you control it? Can it be a separate protocol?

Picking Primitives

- Figure out if you need signatures.
  - Usually depends on if certificates are involved.
  - Ed25519 or NIST P-256
- Use ephemeral Diffie-Hellman for forward secrecy.
  - X25519
- If you don't need signatures, use Triple-DH for authentication
  - Less primitives means less code and less mistakes.
- Use an AEAD for all symmetric crypto.
  - Authenticating a plaintext header as part of the decryption step is 🚀

Recommendations

Opt for protocol designs that lend themselves to secure implementations. The designer should also be an implementer. Redesign things that make the code gross.
This is somewhat tautological.

Keep one joint, and keep it well-oiled. Don’t use cryptographic agility to build backdoors.
Did I really spend all this time talking about export ciphers just to plug the papers you wrote in grad school?

Public-key cryptography is for key exchange and signatures, not encryption. Stop using RSA. Don’t talk about “public-key encryption”.

Use an AEAD. Don’t use static Diffie-Hellman unless you also use ephermeal Diffie-Hellman.

Other Tips

Linearize Your State Transitions. Figure out what path you’re taking as soon as possible, and predefine the only possible orders. Don’t branch.
Example: s2n state machine

Length-Prefixed Vectors. Ideally, everything is fixed length and there is “no parsing”. If not, use length-prefixed vectors, and chunk things off one at a time.
Example: cyryptobyte in Golang

Dogfood. Use your own library. Does it work? How much does it suck to use? Can your server talk to your own client? Can it talk to other implementations? Are the bytes on the wire correct?
Example: Tests for the TLS implementation in Golang

Oops, more tips.

Nonce Management. Figure out what the failure cases of nonce collision is. Do you need 8, 12, or 16-byte nonces? Is your nonce random or a counter?

Replay Attacks. Are you getting help from TCP? What about at the start or resumption of a session? TCP + randomness in handshake + full commitment + no resumption means no replays.

Max Counter. How much data can you send in a single session before you risk birthday attacks?

Formal Verification. Have you formally verified your protocol? If not, could you? If not, why not?

NEW DIRECTIONS IN SECURE TRANSPORTS

UNIVERSITY OF
MICHIGAN

Wireguard

Wireguard is a VPN protocol with only two cryptography-related settings

- Which `X25519` key to use?
- Which `X25519` keys to trust?

There is no cryptographic agility. There is no key distribution. There is no certificate chaining.

Keys can be trivially regenerated in association with an external key management protocol.

Cryptokey routing:

Cryptokey Routing: IP + Key as Routing Table and ACL

Server

```
[Interface]
PrivateKey = yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 51820
[Peer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32, 10.192.124.1/24
[Peer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32, 192.168.0.0/16
```

Client

```
[Interface]
PrivateKey = gI6EdUSYvn8ugXOt8QQD6Yc+JyiZxIhp3GInSWRfWGE=
ListenPort = 21841
[Peer]
PublicKey = HIgo9xNzJMWLKASShiTqIybxZ0U3wGLiUeJ1PKf8ykw=
Endpoint = 192.95.5.69:51820
AllowedIPs = 0.0.0.0/0
```

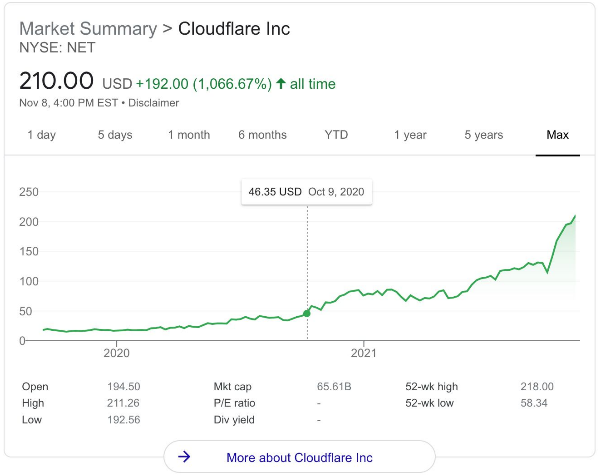
Tailscale

Tailscale is key distribution via SAML and OIDC and automatic route management for Wireguard, used to create an overlay network where each IP address is authenticated via an IdP.

An **overlay network** is a multi-party VPN that supports direct endpoint-to-endpoint communication.

Cloudflare One

Connecting SAML and OIDC to a VPN sounds a lot like “zero-trust”. What happens when you start overlay networks with access control, identity management, policy enforcement, and device management?



Market Summary > Cloudflare Inc
NYSE: NET

210.00 USD +192.00 (1,066.67%) ↑ all time
Nov 8, 4:00 PM EST • Disclaimer

Open	194.50	Mkt cap	65.61B	52-wk high	218.00
High	211.26	P/E ratio	-	52-wk low	58.34
Low	192.56	Div yield	-		

More about Cloudflare Inc

Innovation is still happening.

There are things happening in academia.

Formal Verification, NIST Lightweight Crypto, TLS, Key Transparency

There is money to be made in industry.

Cloudflare One, Tailscale, Nebula, Xaptum, Subspace, Bastion Zero, Hashicorp Boundary

Secure Internet Protocols

DAVID ADRIAN

https://dadrian.io

Zakir's Class

November 8, 2021

UNIVERSITY OF MICHIGAN