AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 2: Real-World Cryptography

2.1: Transport Layer Security

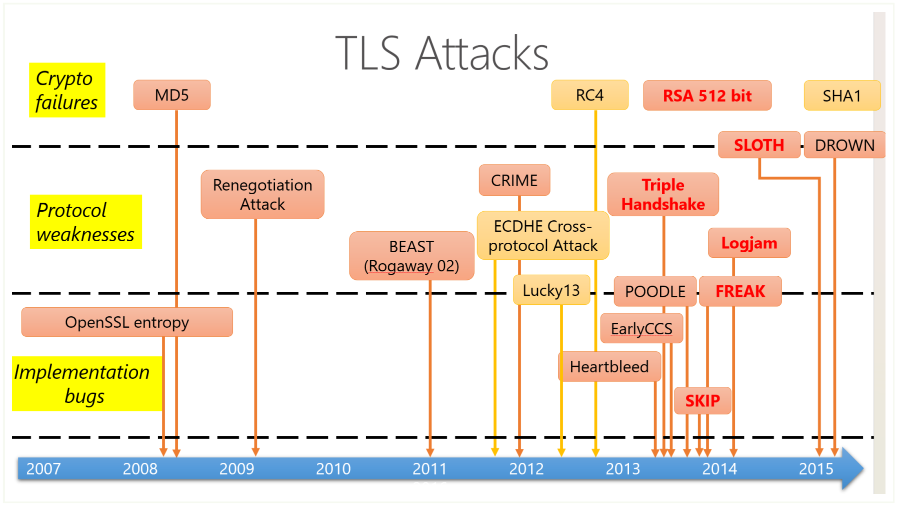
Nadim Kobeissi

https://appliedcryptography.page

Why we’re interested in TLS

- Encrypts all your web traffic (the **S** in HTTP**S**!)
- Great way to introduce **authenticated key exchange**.
- Great way to make you hate certificates and certificate authorities.
- How TLS handles **authentication** and key exchange.
- A comprehensive survey of **TLS 1.2 attacks**:
  - POODLE (Padding Oracle On Downgraded Legacy Encryption)
  - Lucky Thirteen timing attack
  - SMACK (State Machine AttaCK)
  - FREAK (Factoring RSA Export Keys)
  - Logjam attack on Diffie-Hellman
  - And many more...
- How these attacks informed the design of **TLS 1.3**.
- Lessons learned for building secure protocols.

TLS attacks timeline (to 2015)



TLS attacks up until 2015.

TLS isn't very interesting

- Too complicated for no good reason.
- Doesn't accomplish properties as interesting as those of more modern protocols, or even Signal.
- Certificate authorities.
- Very important however to understand, basically a golden door into the world of protocols.

...and yet...

- Web browsing (HTTPS)
- Email (IMAPS, POP3S, SMTPS)
- Instant messaging (WhatsApp, Telegram)
- Video conferencing (Zoom, Teams)
- Online banking
- Social media (Facebook, Twitter)
- Cloud storage (Dropbox, Google Drive)
- VPN connections
- Mobile app communication
- Medical devices
- Streaming services (Netflix, Spotify)
- IoT device communication
- Remote desktop protocols
- Voice over IP (VoIP)
- Software updates
- Database connections
- Git repositories (GitHub, GitLab)
- Real-time communication (WebRTC)
- Cryptocurrency wallets
- Smart home devices

Section 1

Transport Layer Security

Primary applications that drove TLS development

- **E-commerce websites**
  - Credit card numbers
  - Personal information
  - Purchase history
- **Online banking**
  - Account credentials
  - Financial transactions
  - Sensitive financial data
- **General web browsing**
  - User credentials
  - Private communications
  - Personal data protection

Passive vs Active Attackers

Passive Attacker

- **Eavesdropping only**
- Observes network traffic
- Records all messages
- Cannot modify or inject messages
- Examples:
  - ISP logging traffic
  - Government surveillance
  - WiFi sniffing

Active Attacker

- **Can modify/inject messages**
- Intercepts and alters traffic
- Can impersonate parties
- Controls network routing
- Examples:
  - Man-in-the-middle attacks
  - Malicious WiFi hotspots
  - Compromised routers

Protocol Design Impact

Identity protection works against passive attackers, but active attackers can force identity revelation through protocol manipulation

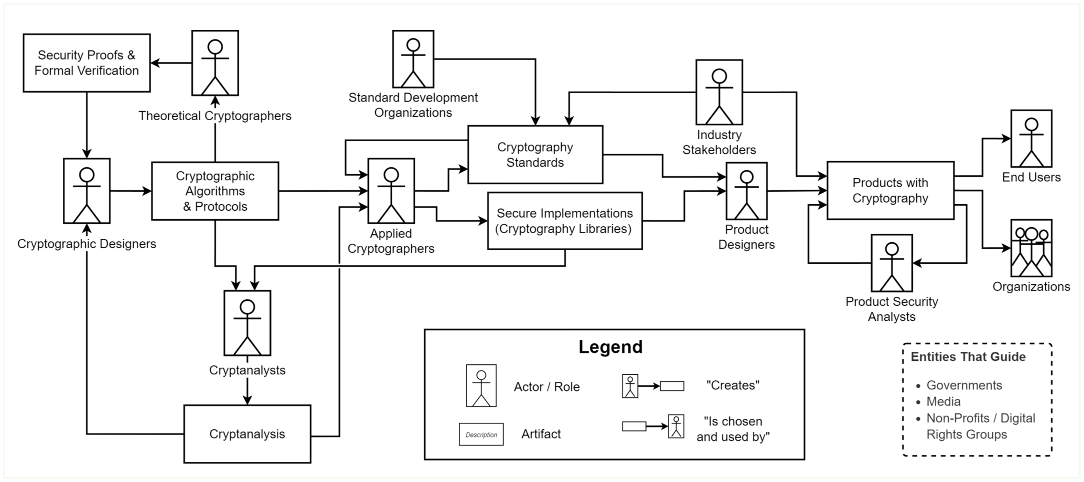
Key security goal: Defeating man-in-the-middle attacks

- **MITM attack scenario:**
  1. Attacker intercepts encrypted traffic.
  2. Decrypts the traffic to read content.
  3. Re-encrypts and forwards to destination.
- **How TLS defeats MITM:**
  - Server authentication using **certificates**.
  - Trusted **certificate authorities** (CAs).
  - Optional client authentication.
- Without proper authentication, encryption alone is **not enough**!

Four requirements for wide TLS adoption

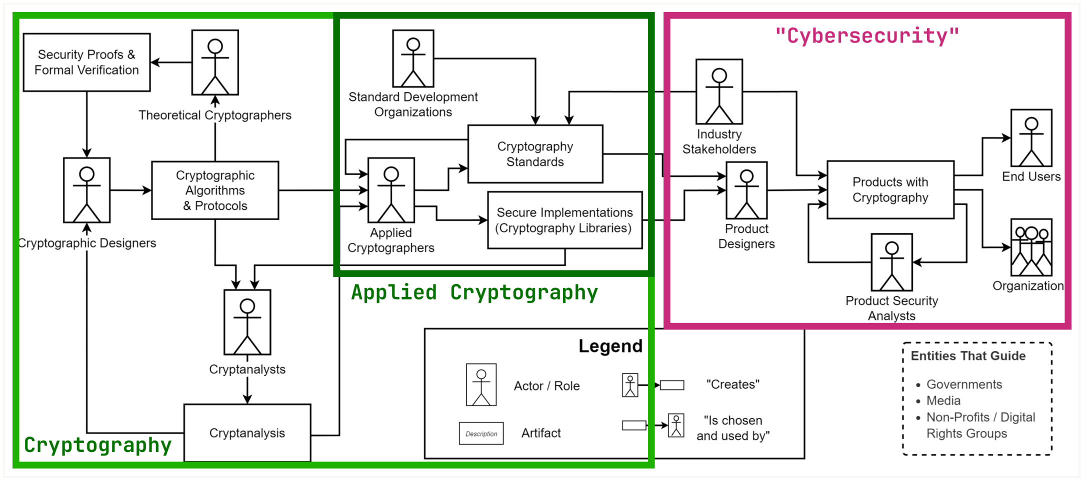
- **Efficiency**
  - Minimize performance penalty vs. unencrypted connections.
- **Interoperability**
  - Work on any hardware and operating system.
- **Extensibility**
  - Support additional features and algorithms.
- **Versatility**
  - Not bound to specific applications.

Notice how we're mentioning design requirements now?



Fischer et al., The Challenges of Bringing Cryptography from Research Papers to Products: Results from an Interview Study with Experts, USENIX Security 2024

Notice how we're mentioning design requirements now?



Fischer et al., The Challenges of Bringing Cryptography from Research Papers to Products: Results from an Interview Study with Experts, USENIX Security 2024

Efficiency and interoperability

Efficiency matters for:

- **Servers**
  - Reduce hardware costs
  - Handle more connections
- **Clients**
  - Avoid perceptible delays
  - Preserve battery life

Interoperability ensures:

- Works across different:
  - Hardware platforms
  - Operating systems
  - Software implementations
- Universal compatibility
- No vendor lock-in

Extensibility and versatility

Extensibility allows:

- Adding new cryptographic algorithms
- Supporting new features
- Adapting to security threats
- Protocol evolution over time

Versatility means:

- Application-agnostic design
- Like TCP: doesn’t care about upper layers
- Can secure any application protocol
- Reusable security infrastructure

The confusing SSL/TLS naming

- **1995**: Netscape develops SSL (Secure Sockets Layer).
- **SSL 2.0 and SSL 3.0**: Both had serious security flaws.
- **Confusing terminology**: People still call TLS “SSL”.
  - Even security experts do this!
  - “SSL certificate” really means “TLS certificate”
- TLS = Transport Layer Security (SSL’s successor)

TLS version evolution

- **TLS 1.0 (1999)**
  - Least secure TLS version.
  - Still better than SSL 3.0.
- **TLS 1.1 (2006)**
  - Better, but includes weak algorithms.
- **TLS 1.2 (2008)**
  - Much better, but very complex.
  - High security only if configured correctly.
  - Supports vulnerable features (e.g., AES-CBC with padding oracles).

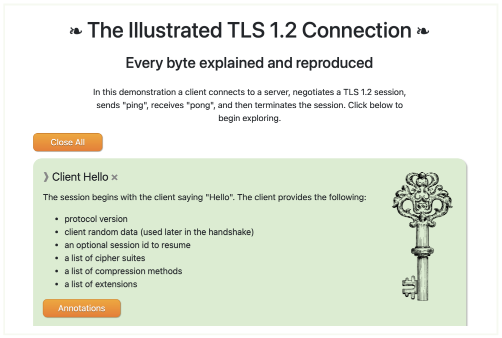
TLS 1.3: The great cleanup

- **Problem**: TLS 1.2 inherited decades of legacy features.
  - Suboptimal security and performance.
  - Complex and error-prone configurations.
  - High risk of implementation bugs.
- **Solution**: Complete redesign for TLS 1.3.
  - Kept only the good parts.
  - Added modern security features.
  - Simplified the bloated design.
- **Result**: TLS 1.3 is more secure, efficient, and simpler.
- TLS 1.3 = “mature TLS”.

Fantastic resource on understanding TLS

And secure channel protocol design in general

- The Illustrated TLS 1.2 connection: `https://tls12.xargs.org`
- The Illustrated TLS 1.3 connection: `https://tls13.xargs.org`



TLS architecture: Two main protocols

- **Handshake Protocol**
  - Determines secret keys shared between client and server.
  - Handles authentication and key exchange.
  - Runs once at the beginning of a connection.
- **Record Protocol**
  - Describes how to use established keys to protect data.
  - Processes data packets called **records**.
  - Defines packet format for encapsulating higher-layer data.
- Think of it as: handshake = setup, record = ongoing protection.

Cipher Suites: The building blocks of TLS security

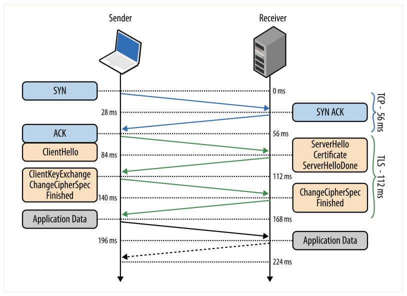
- **What is a cipher suite?**
  - A combination of cryptographic algorithms used together
  - Defines exactly how data will be secured
  - Like a recipe: specifies all ingredients for secure communication
- **Four components of a cipher suite**:
  1. **Key exchange algorithm**: How to establish shared keys (RSA, ECDHE, DHE)
  2. **Authentication algorithm**: How to verify identity (RSA, ECDSA)
  3. **Bulk encryption algorithm**: How to encrypt data (AES, ChaCha20)
  4. **MAC algorithm**: How to ensure integrity (SHA256, SHA384, Poly1305)
- **Example**: `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
  - `ECDHE`: Elliptic Curve Diffie-Hellman Ephemeral (key exchange)
  - `RSA`: RSA signatures (authentication)
  - `AES_128_GCM`: AES with 128-bit keys in GCM mode (encryption + MAC)
  - `SHA256`: SHA-256 for handshake integrity

Common cipher suites in practice

- **TLS 1.2 cipher suites** (verbose naming):
  - `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
  - `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
  - `TLS_RSA_WITH_AES_128_CBC_SHA256`
  - `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`
- **TLS 1.3 cipher suites** (simplified naming):
  - `TLS_AES_128_GCM_SHA256`
  - `TLS_AES_256_GCM_SHA384`
  - `TLS_CHACHA20_POLY1305_SHA256`
- **Why TLS 1.3 names are shorter**:
  - Key exchange is always (EC)DHE (forward secrecy mandatory)
  - Authentication tied to certificate type
  - Only specifies symmetric crypto algorithms
- **Cipher suite negotiation**: Client proposes, server chooses

The TLS handshake: Basic flow

- **Step 1**: Client initiates secure connection.
    - Sends `ClientHello` message.
    - Includes supported ciphers and other parameters.
- **Step 2**: Server responds.
    - Checks client's message and parameters.
    - Responds with `ServerHello` message.
    - Selects cipher suite and provides certificate.
- **Step 3**: Key establishment.
    - Both parties process each other's messages.
    - Establish session keys for encryption.
- **Result**: Ready to exchange encrypted data!



TLS handshake overview.

Critical step: Certificate validation

- **The crux of TLS security**: Server authenticates itself to client
- **What is a certificate?**
  - A public key + signature of that key
    - Encoded in an insanely asinine byte format
  - Associated information (domain name, organization, etc.)
  - Essentially says: “I am google.com, and my public key is [key]”
- **Certificate validation process**:
  1. Browser receives certificate from server
  2. Verifies the certificate’s signature
  3. If signature is valid →certificate and public key are trusted
  4. Browser proceeds with connection

Certificate Authorities: The trust foundation

- **Problem**: How does the browser know which public key to use for verification?
- **Solution**: Certificate Authorities (CAs)
  - Public keys hardcoded in your browser/OS.
  - Trusted organizations that issue certificates.
  - Act as trusted third parties.
- **CA's role**:
  - Verify that public keys belong to claimed entities.
  - Sign certificates with their private keys.
  - Protect their private keys from compromise.
- **Without CAs**: No way to distinguish legitimate servers from MITM attackers!

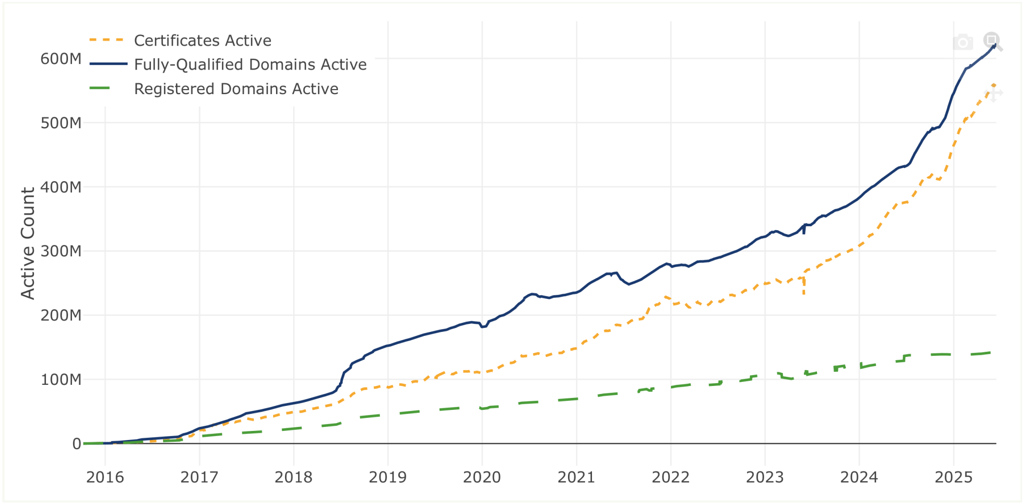
Traditional CA validation nightmare

- **Getting a certificate before 2015**:
  - Contact a Certificate Authority (Verisign, Thawte, etc.).
  - Pay $50-300+ per certificate per year.
    - Completely arbitrary extortionate amounts.
  - Manual validation process takes days/weeks.
- **Domain Validation (DV) process**:
  1. Submit CSR (Certificate Signing Request).
  2. CA sends email to `admin@domain.com`.
  3. Click verification link in email.
  4. Wait for manual review and approval.
  5. Download and install certificate.
- **Extended Validation (EV) certificates**:
  - Even more expensive ($150-1000+/year).
  - Weeks of back-and-forth communication.

Let’s Encrypt: revolution in TLS certificates (2015)

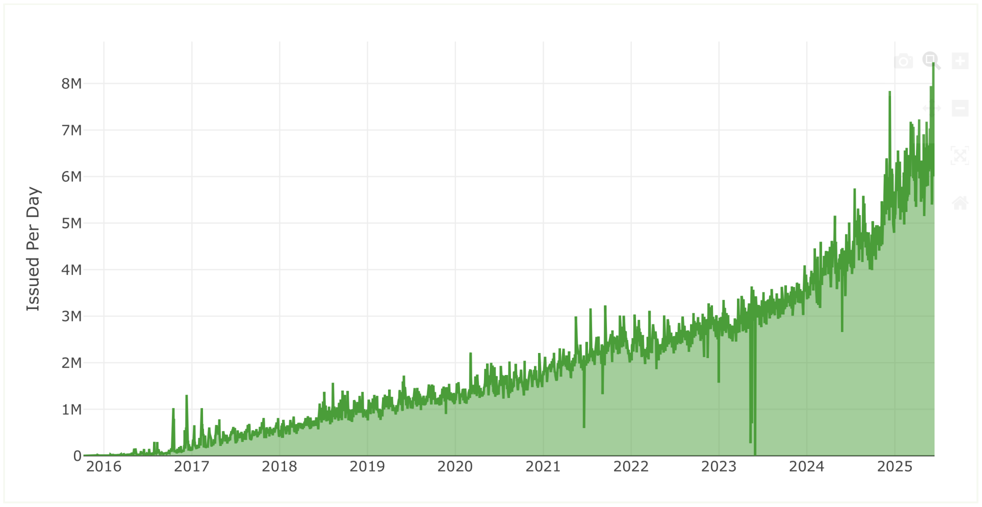
- **Game-changing**:
  - **Free** certificates for everyone.
  - **Automated** issuance and renewal.
  - **90-day** certificate lifetime (encourages automation).
  - Open source, non-profit initiative.
- **ACME protocol** (Automated Certificate Management Environment):
  - Domain validation via HTTP or DNS challenges.
  - Prove control of domain programmatically.
  - Certificate issued in seconds, not days.
  - Automatic renewal before expiration.
- **Impact on the web**:
  - HTTPS adoption jumped from 40% to 90%+ of web traffic.
  - Eliminated cost barrier for small websites.
  - Made “HTTPS everywhere” a reality.
- **Philosophy**: Encryption should be the default, not a luxury.

Let's Encrypt: active certificates



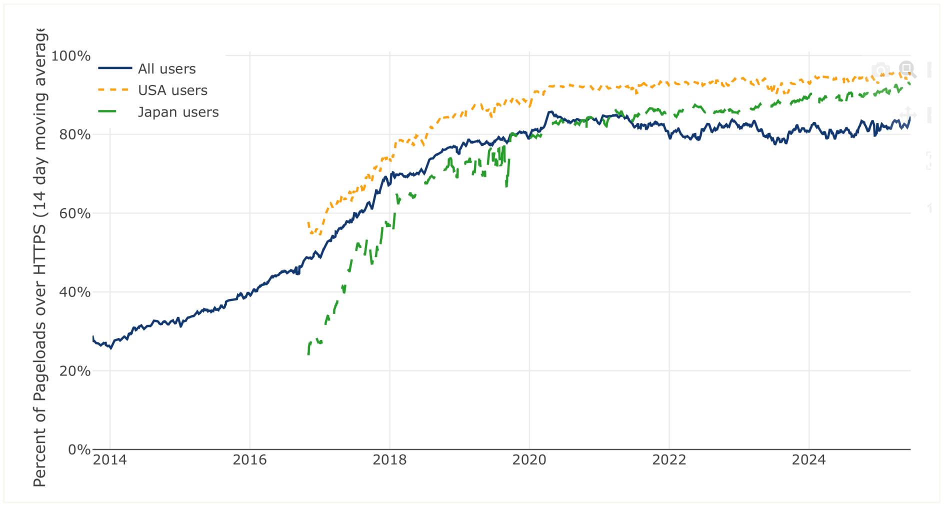
Source: Let's Encrypt

Let’s Encrypt: certificates issued



Source: Let’s Encrypt

Percentage of HTTPS traffic in Firefox



Source: Let's Encrypt

Certificate chains in practice

- **Real-world example**: Connecting to `www.google.com`
- **Certificate chain structure**:
  - **Certificate 0**: `www.google.com`'s certificate.
  - **Certificate 1**: Intermediate CA (Google Trust Services).
  - **Certificate 2**: Root CA (GlobalSign).
- **Verification process**:
  1. Check Google's signature on Certificate 0.
  2. Check GlobalSign's signature on Certificate 1.
  3. Trust GlobalSign's root certificate (pre-installed).
- **Chain of trust**: Each certificate vouches for the next.

Exploring certificates with OpenSSL

- **Connect and view certificate**:
  - `openssl s_client -connect www.google.com:443`
  - Shows certificate chain and raw certificate data
- **Parse certificate details**:
  - `openssl x509 -text -noout`
  - Reveals subject, issuer, validity dates, algorithms
- **Certificate markers**:
  - `s:` = subject (who the certificate is for)
  - `i:` = issuer (who signed the certificate)
- Try this at home! Great way to understand the certificate ecosystem.

TLS Record Protocol: The data transport layer

- **What is the Record Protocol?**
  - Transport protocol for all TLS data.
  - Agnostic to the meaning of transported data.
  - Makes TLS suitable for any application.
- **Two main phases**:
  1. **During handshake**: Carries handshake messages.
  2. **After handshake**: Carries encrypted application data.
- **Key insight**: Like TCP, doesn't care about upper layers!

TLS Record: Basic structure

- **TLS Record = chunk of data ≤ 16KB**
- **Simple header structure**:
  - Only 3 fields (vs. 14 in IPv4, 13 in TCP)
  - 5-byte header + payload
- **Visual representation**:

ContentType	ProtocolVersion	Length	Payload
1 byte	2 bytes	2 bytes	≤ 16KB

TLS Record fields breakdown

- **Byte 1 - ContentType**:
  - 22 = Handshake data
  - 23 = Encrypted application data
  - 21 = Alerts (error messages)
- **Bytes 2-3 - ProtocolVersion**:
  - Always `03 01` (historical reasons)
  - Same across TLS versions (confusing!)
- **Bytes 4-5 - Length**:
  - 16-bit integer encoding payload length
  - Maximum: 2^{14} bytes = 16KB

ContentType: What's in this record?

Value	Name	Contains
21	Alert	Error messages, warnings
22	Handshake	ClientHello, ServerHello, Certificate, etc.
23	Application Data	Encrypted user data (HTTP, email, etc.)

- **Key point**: ContentType tells receiver how to process the payload.
- **During handshake**: Mostly type 22 records.
- **After handshake**: Mostly type 23 records.

Encrypted records (ContentType = 23)

- **When ContentType = 23**:
  - Payload is encrypted and authenticated.
  - Contains: ciphertext + authentication tag.
- **How does receiver know how to decrypt?**
  - Cipher and key established during handshake.
  - “Magic of TLS”: if you receive encrypted data, you already have the key!
- **Decryption process**:
  1. Verify authentication tag.
  2. Decrypt ciphertext.
  3. Process decrypted application data.

Nonces: Ensuring unique encryption

- **Problem**: Each record needs a unique nonce for encryption.
- **TLS solution**: Derive nonces from sequence numbers.
  - Each party maintains 64-bit sequence number.
  - Incremented for each new record.
  - Starts at 0, goes 1, 2, 3, ...
- **Nonce derivation**:
  - Client: `sequence_number` ⊕ `client_write_iv`
  - Server: `sequence_number` ⊕ `server_write_iv`
- **No nonce reuse**: Different IVs and keys per direction.

Example: Sequence numbers in action

Client sending records:

Record #	Sequence Number
1st record	0
2nd record	1
3rd record	2

Client receiving records:

Record #	Sequence Number
1st record	0
2nd record	1
3rd record	2

- **Safe to reuse numbers**: Different keys and IVs per direction!

Zero padding example

Without padding:

- Short message → small ciphertext
- Long message → large ciphertext
- Attacker learns message sizes

With padding:

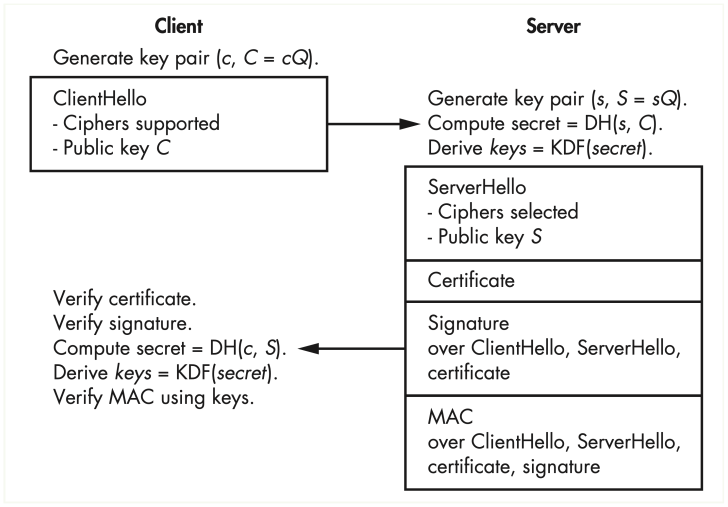
- Short message + padding → large ciphertext
- Long message + padding → large ciphertext
- All messages look the same size!

Trade-off: Security vs. bandwidth efficiency

TLS: the handshake protocol

- **The crux of TLS**: Process to establish shared secret keys.
- **Goal**: Initiate secure communications between client and server.
- **Key outcome**: Both parties agree on:
  - TLS version to use
  - Cipher suite for encryption
  - Shared secret keys for protection
- **Interoperability requirement**: Any TLS 1.3 client must work with any TLS 1.3 server.
  - Achieved through standardized message formats.
  - Works regardless of implementation or programming language.

TLS: the handshake protocol



TLS 1.3 handshake protocol. Source: Serious Cryptography

Client vs. Server: Different roles in the handshake

Client's role:

- **Proposes** configurations
- Lists supported TLS versions
- Lists supported cipher suites
- Orders preferences (most preferred first)
- Generates Diffie-Hellman key pair

Server's role:

- **Chooses** final configuration
- Selects TLS version to use
- Picks cipher suite from client's list
- Should follow client's preferences
- Responds with its own DH public key

Think of it like ordering at a restaurant:

Client: "Here's what I like..." Server: "We'll go with this option."

ClientHello: “I want to establish a TLS connection”

- **Client’s opening message to server**
- **Key contents**:
  - List of supported cipher suites (in order of preference)
  - Diffie-Hellman public key (generated just for this session!)
  - 32-byte random value
  - Optional extensions and parameters
- **Critical security detail**: DH private key stays with client.
- **Format requirement**: Must follow exact byte format from TLS 1.3 spec.
  - Ensures any server can parse any client’s message.

ServerHello: “Here’s what we’ll use and who I am”

- **Server’s response**: Packed with crucial information!
- **Configuration decisions**:
  - Selected cipher suite (from client’s list)
  - Server’s Diffie-Hellman public key
  - 32-byte random value
- **Authentication materials**:
  - Certificate (contains server’s public key)
  - Signature of ClientHello + ServerHello contents
  - MAC of all the above information
- **Crypto magic**: MAC uses key derived from DH shared secret!

Client verification: “Can I trust this server?”

- **When client receives ServerHello**:
- **Step 1**: Certificate validation
  - Check certificate chain to trusted CA
  - Verify certificate hasn’t expired
  - Confirm certificate matches domain name
- **Step 2**: Signature verification
  - Use certificate’s public key to verify signature
  - Ensures server controls the private key
- **Step 3**: Derive shared secrets
  - Compute DH shared secret: $g^{ab} \mod p$
  - Derive symmetric keys from shared secret
- **Step 4**: MAC verification
  - Verify MAC using derived symmetric key

All checks pass: Ready for encrypted communication!

After successful verification:

- Client is confident it's talking to the legitimate server.
- Both parties have the same shared secret keys.
- Ready to send encrypted application data!
- **Security guarantees achieved**:
  - **Confidentiality**: Traffic is encrypted
  - **Integrity**: Traffic is authenticated
  - **Authentication**: Server identity verified

Real-world example: Visiting `aub.edu.lb`

Step 1: Browser sends ClientHello

- Lists supported ciphers
- Includes DH public key
- Adds random nonce

Step 2: Server responds

- Sends ServerHello
- Provides certificate for "`aub.edu.lb`"
- Includes signature and MAC

Step 3: Browser verification

- Certificate validated using browser's built-in CA certificates
- Must be signed by trusted certificate authority
- Certificate must match domain name: "`aub.edu.lb`"
- All cryptographic checks must pass

Step 4: Browser requests initial page over encrypted channel!

Security guarantees after successful TLS handshake

- **All communications are encrypted and authenticated**
- **What an eavesdropper can see**:
  - Client IP address
  - Server IP address
  - Encrypted content (but can't read it!)
  - Traffic patterns and timing
- **What an eavesdropper CANNOT do**:
  - Read the underlying plaintext
  - Modify messages without detection
  - Impersonate either party
- **Authentication guarantee**: If messages are tampered with, receiving party will detect it!
- **Bottom line**: Enough security for most applications.

Forward Secrecy: Protecting past communications

- **The problem**: What if a server's private key is compromised?
  - Attacker could decrypt **all past** TLS sessions.
  - Years of stored encrypted traffic becomes readable.
  - Example: NSA's alleged collection of encrypted internet traffic.
- **Forward secrecy (Perfect Forward Secrecy - PFS)**:
  - Ensures past sessions remain secure even if long-term keys are compromised.
  - Each session uses unique, **ephemeral keys**.
    - *ephemeral: "lasting a very short time"*
  - Session keys are deleted after use.
- **Security guarantee**: "Even if you compromise me today, you can't read yesterday's traffic."
- **Critical for**: Journalists, activists, whistleblowers, ordinary citizens

How TLS achieves forward secrecy

- **Ephemeral Diffie-Hellman key exchange**:
  - Server generates fresh DH key pair for each session.
  - Client generates fresh DH key pair for each session.
  - Shared secret computed: $g^{ab} \bmod p$ (then deleted!).
- **Key lifecycle**:
  1. Generate ephemeral keys for handshake.
  2. Derive session keys from ephemeral shared secret.
  3. **Delete ephemeral private keys immediately**.
  4. Use session keys for encrypted communication.
  5. Delete session keys when connection ends.
- **Server's long-term private key**: Only used to **sign** the handshake
  - Not used to derive session keys directly.
  - Compromising it doesn't reveal past session keys.
- **Result**: Each TLS session is cryptographically independent.

How things can go wrong with TLS

- **Common TLS failure scenarios:**
  - Even with the most secure ciphers, TLS can be compromised
  - Security relies on assumptions about honest behavior
- **Key assumption**: All three parties behave honestly
  - Client
  - Server
  - Certificate Authority (CA)
- **Reality check**: What if one party is compromised?
- **Implementation matters**: Poor TLS implementations create vulnerabilities

Compromised Certificate Authority

- **Root CAs**: Organizations that browsers trust to validate certificates
- **Normal process**:
  - CA verifies legitimacy of certificate owner.
  - CA signs certificate with their private key.
  - Browser trusts CA's signature.
- **Attack scenario**: CA's private key is compromised
  - Attacker can create certificates for **any domain**.
  - No approval needed from actual domain owner.
  - Can impersonate legitimate servers.
- **Attack capabilities**:
  - Create fake certificates for `google.com`, `instagram.com`, etc.
  - Intercept user credentials and communications.
  - Man-in-the-middle attacks become trivial.

Real-world example: DigiNotar

- **What happened**:
  - Dutch certificate authority DigiNotar was hacked.
  - Attackers gained access to CA's private keys.
  - Created fake certificates for Google services.
- **Impact**:
  - Users unknowingly connected to malicious servers.
  - Credentials and communications intercepted.
  - Trust in entire CA system questioned.
- **Aftermath**:
  - DigiNotar went bankrupt.
  - All major browsers removed DigiNotar certificates.
  - Led to improved CA monitoring and transparency.
- **Lesson**: CAs are high-value targets for attackers!

Real-world example: Kazakhstan government certificate

- **2015**: Kazakhstan government creates "national security certificate"
  - Root certificate that could enable MITM attacks.
  - Would allow government to intercept HTTPS traffic.
  - Required manual installation on users' devices.
- **July 2019**: Government mandates certificate installation
  - ISPs instructed users to install "Qaznet Trust Certificate".
  - Issued by state CA: Qaznet Trust Network.
  - Initial targets: Google, Facebook, Twitter.
- **August 2019**: Browser vendors fight back
  - Mozilla (Firefox) and Google (Chrome) block the certificate.
  - Apple (Safari) joins the blocking effort.
  - Microsoft reiterates certificate not in trusted root store.
- **December 2020**: Government tries again, browsers block again

Kazakhstan case: Lessons for TLS security

- **Government-level MITM attempts are real**
  - Nation-states can create sophisticated infrastructure.
  - Legal pressure on ISPs and users.
  - "National security" justifications.
- **Browser vendors as guardians**:
  - Coordinate to protect users from malicious CAs.
  - Can override user certificate installations.
  - Technical measures against political pressure.
- **Certificate transparency importance**:
  - Public logs make rogue certificates detectable.
  - Community monitoring of CA behavior.
  - Enables coordinated responses to threats.
- **Bottom line**: Even governments can't easily break modern TLS

Certificate Transparency

- **The problem**: CAs can issue certificates without anyone knowing.
  - Rogue certificates for targeted attacks.
  - Compromised CAs issuing malicious certificates.
  - No way to detect misbehavior after the fact.
- **Certificate Transparency (CT) solution** (2013):
  - CAs must submit all certificates to public logs.
  - Logs are cryptographically append-only.
  - Anyone can monitor logs for suspicious certificates.
- **How it works**:
  1. CA issues certificate and submits to CT logs.
  2. Log returns Signed Certificate Timestamp (SCT).
  3. Certificate includes SCT as proof of logging.
  4. Browsers reject certificates without valid SCTs.
- **Result**: CA misbehavior becomes publicly detectable!

Example: How Let's Encrypt uses CT

- **The Static CT API**: Evolution of Certificate Transparency.
  - Logs represented as simple flat files ("tiles").
  - Download log data just like downloading files.
  - CDN-friendly, cheaper to operate.
- **Key advantages**:
  - 10x+ cheaper to operate ($10k/year vs traditional logs).
  - More reliable (simpler architecture).
  - Easier to download and share data.
- **Sunlight**:
  - Open source CT implementation used by Let's Encrypt.[a]
  - `https://sunlight.dev`
- **Status**: Chrome and Safari now accepting Static CT API logs!

[a] https://letsencrypt.org/2025/06/11/reflections-on-a-year-of-sunlight/

Critical examination: Are browser vendors truly neutral?

- **Browser vendors as gatekeepers**:
  - Google (Chrome), Mozilla (Firefox), Apple (Safari), Microsoft (Edge).
  - Unilateral power to accept/reject certificates.
  - Who watches the watchers?
- **Political and economic pressures**:
  - Companies have business interests and government relations.
  - What if a government pressures Google/Apple directly?
  - Corporate decisions affecting global internet security.
- **The CA model's fundamental problems**:
  - **Single point of failure**: Any CA can issue certificates for any domain.
  - **Asymmetric trust**: Must trust **all** CAs, not just one.
  - **Centralized control**: Small number of entities control global trust.
  - **Economic incentives**: CAs profit from issuing more certificates.
- **Question**: Is this the best we can do for global internet security?

Thinking beyond: Decentralized trust alternatives

- **Why consider alternatives?**
  - Reduce single points of failure.
  - Eliminate centralized gatekeepers.
  - Increase transparency and auditability.
- **Blockchain-based certificate systems**:
  - Certificates recorded on public blockchain.
  - Cryptographic proof of certificate history.
  - Examples: Namecoin, Ethereum Name Service (ENS)
  - **Trade-offs**: Scalability, energy consumption, governance.
- **Web of Trust models**:
  - Users vouch for each other's identities (like PGP).
  - Decentralized reputation systems.
  - **Trade-offs**: User complexity, bootstrap problem
- **DNS-based alternatives**: DANE (DNS-based Authentication of Named Entities)
- **Your turn**: What other models could work? What are the trade-offs?

Compromised Server

- **Worst-case scenario**: Server is fully controlled by attacker
- **What the attacker gains**:
  - Session keys (server is TLS termination point)
  - All transmitted data **before** encryption
  - All received data **after** decryption
  - Server's private key
- **Attack capabilities**:
  - Read all user communications in plaintext
  - Impersonate the legitimate server
  - Use private key to create malicious servers
- **Bottom line**: TLS won't save you if the server is compromised!

Server compromise: Mitigation strategies

- **Good news**: High-profile services are well-protected
  - Gmail, iCloud, major banks
  - Multiple layers of security
- **Hardware Security Modules (HSMs)**:
  - Store private keys in separate, tamper-resistant hardware
  - Even if server is compromised, keys may remain safe
- **Key Management Systems (KMS)**:
  - Centralized key storage and management
  - Limits exposure of private keys
- **More common threats**: Web application vulnerabilities
  - SQL injection, cross-site scripting (XSS)
  - Carried out **over** legitimate TLS connections
  - Independent of TLS security

Compromised Client

- **Attack scenario**: Browser or client application is compromised
- **What the attacker gains**:
  - Session keys from compromised client
  - All decrypted data visible to client
  - Ability to modify client behavior
- **Advanced attack**: Installing rogue CA certificates
  - Add malicious CA to client's trusted store
  - Client silently accepts invalid certificates
  - Enables seamless man-in-the-middle attacks
- **Scope difference**:
  - Compromised CA/server: affects **all** clients
  - Compromised client: affects **only that** client

Client compromise: Attack techniques

- **Malware installation**:
  - Trojans, viruses, browser plugins
  - Can intercept data before encryption
- **Certificate store manipulation**:
  - Add attacker's CA certificate to trusted store
  - Modify certificate validation logic
- **Browser hijacking**:
  - Redirect traffic through attacker's proxy
  - Modify DNS settings
- **Protection strategies**:
  - Keep browsers and OS updated
  - Use reputable antivirus software
  - Certificate pinning (for developers)
  - Monitor certificate store changes

TLS security: A chain is only as strong as its weakest link

Three points of failure in the TLS trust model:

Certificate Authority

- Must protect private keys
- Must verify identities correctly
- Must not issue rogue certificates

Server

- Must protect private keys
- Must maintain secure infrastructure
- Must handle session keys safely

Client

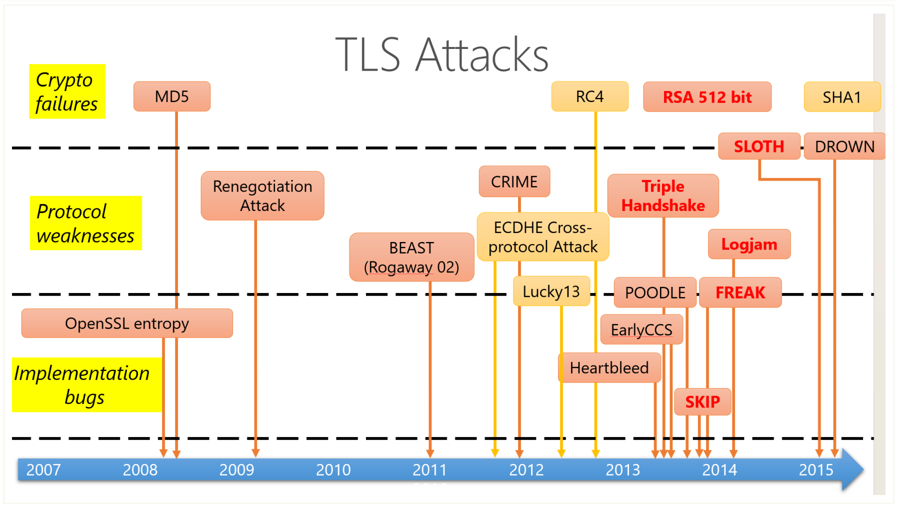
- Must validate certificates properly
- Must protect trusted CA store
- Must remain malware-free

Compromise any one component → TLS security fails!

Section 2

Attacks on TLS

TLS attacks timeline (to 2015)



TLS attacks up until 2015.

Lucky Thirteen (2013)

- **Target**: TLS's CBC (Cipher Block Chaining) mode with HMAC
- **The vulnerability**: Timing differences in MAC verification[a]
  - TLS 1.0-1.2 used MAC-then-encrypt with CBC mode
  - Padding oracle attacks exploit timing differences
  - Different MAC verification procedures for valid vs. invalid padding
- **Attack mechanism**:
  1. Attacker modifies ciphertext to create invalid padding
  2. Measures server response time
  3. Timing differences reveal padding validity
  4. Uses timing to guess plaintext byte-by-byte
- **Why "Lucky Thirteen"?** Attack requires exactly 13 bytes of padding

---

[a] https://appliedcryptography.page/papers/#lucky-thirteen

Lucky Thirteen: Technical details

- **CBC padding in TLS**:
  - Messages padded to block boundary (16 bytes for AES)
  - Padding bytes contain length of padding
  - Example: `...data|05|05|05|05|05|05` (5 bytes padding)
- **The timing leak**:
  - **Valid padding**: MAC checked on message after padding removal
  - **Invalid padding**: TLS spec says to check MAC as if padding had zero-length
  - Different amounts of data processed creates timing difference
- **Statistical attack**:
  - Send thousands of modified ciphertexts
  - Measure response times
  - Statistical analysis reveals timing patterns
- **Result**: Can decrypt HTTPS cookies, passwords, session tokens

Lucky Thirteen: Real-world impact

- **Affected systems**: All TLS 1.0-1.2 implementations using CBC
  - OpenSSL, NSS, GnuTLS, SChannel
  - Millions of web servers worldwide
- **Practical exploitation**:
  - Requires man-in-the-middle position
  - Can extract secrets from encrypted sessions
  - Especially dangerous on shared networks (Wi-Fi)
- **Mitigation attempts**:
  - Constant-time implementations (hard to get right)
  - Prefer AEAD ciphers (AES-GCM)
  - Ultimately: abandon CBC mode entirely
- **Legacy**: Demonstrated fundamental flaws in MAC-then-encrypt

POODLE (2014): Downgrade attacks strike

- **Full name**: Padding Oracle On Downgraded Legacy Encryption[a]
- **Target**: SSL 3.0 (ancient protocol from 1996)
- **The setup**:
  - Browsers support SSL 3.0 for "compatibility"
  - Attacker forces downgrade from TLS to SSL 3.0
  - SSL 3.0 has weaker padding validation
- **Attack flow**:
  1. Client attempts TLS 1.2 connection
  2. Attacker blocks/corrupts TLS handshake
  3. Client "gracefully" falls back to SSL 3.0
  4. Attacker exploits SSL 3.0 padding oracle
- **Discovered by**: Google Security Team

[a] https://appliedcryptography.page/papers/#google-poodle

POODLE: The padding oracle vulnerability

- **SSL 3.0 padding flaw**:
  - Padding bytes can contain **any values**
  - Only padding length is validated
  - Unlike TLS, which requires specific padding patterns
- **Attack technique**:
  - Replace last block of ciphertext with target block
  - If padding is valid, server processes message
  - If padding is invalid, server returns error
  - Use oracle to guess plaintext bytes
- **Efficiency**:
  - Extract one byte per 256 requests (on average)
  - Much faster than previous padding oracle attacks
- **Practical impact**: Steal HTTP cookies, session tokens

POODLE: Industry response and lessons

- **Immediate response**:
  - Major browsers disabled SSL 3.0 support
  - Server administrators configured to reject SSL 3.0
  - “Fallback SCSV” mechanism introduced
- **Fallback SCSV**: Cryptographic downgrade protection
  - Client signals highest supported version
  - Server detects and rejects artificial downgrades
  - Prevents attacker-induced fallbacks
- **Broader lessons**:
  - Backward compatibility creates security risks
  - Legacy protocols should be completely removed
  - Graceful degradation can be graceful exploitation
- **Long-term impact**: Accelerated retirement of old protocols

Triple Handshakes and Cookie Cutters (2014)

- **Discovered by**: Inria Prosecco team (future TLS 1.3 verifiers!)
- **Core problem**: TLS handshake can be **resumed** with different certificates[a]
    - Client connects to Server A, establishes session
    - Session can be resumed with Server B using different certificate
    - Client may not notice certificate change
- **Attack scenario**:
    - Attacker has certificate for `evil.com`
    - Tricks client into resuming session with `good.com`
    - Client thinks it's talking to `good.com`
    - Actually talking to attacker with `evil.com` certificate
- **Impact**: Breaks TLS authentication guarantees

[a] https://appliedcryptography.page/papers/#triple-handshakes

Triple Handshakes: The technical attack

- **Session resumption vulnerability**:
  - TLS allows resuming sessions with different certificates
  - Session keys remain the same
  - Client authentication context gets confused
- **“Triple handshake” attack**:
  1. Handshake 1: Client ← Attacker (using evil certificate)
  2. Handshake 2: Attacker ← Server (using good certificate)
  3. Handshake 3: Client ← Server (resumed session, confused identity)
- **Result**: Client sends sensitive data to attacker
- **Renegotiation attacks**: Similar issues with TLS renegotiation
- **Cookie cutter**: Attacker can splice different handshakes together

Triple Handshakes: Fixes and prevention

- **Extended Master Secret (RFC 7627)**:
  - Bind session keys to handshake transcript
  - Prevents session resumption with different handshakes
  - Master secret includes hash of all handshake messages
- **Renegotiation Indication Extension**:
  - Cryptographically bind renegotiated connections
  - Prevents injection of malicious handshakes
- **TLS 1.3 solution**:
  - Completely removes renegotiation
  - Simplified session resumption with PSK
  - Cannot resume with different certificates
- **Significance**: Showed TLS state machine was more complex than realized

Heartbleed (2014): The bug that broke the internet

- **Not a protocol flaw**: Implementation bug in OpenSSL[a]
- **The vulnerability**: Buffer over-read in heartbeat extension
  - Heartbeat: "keep-alive" mechanism for TLS
  - Client sends data + length field
  - Server echoes data back
- **The bug**: No bounds checking on length field
  - Send 1 byte of data, claim it's 64KB
  - Server copies 64KB from memory
  - Returns server's memory contents to attacker
- **Impact**: Arbitrary memory disclosure

[a] https://appliedcryptography.page/papers/#matter-heartbleed

Heartbleed: What attackers could steal

- **Server's private keys**:
  - TLS private keys used for authentication
  - Allows impersonation of legitimate servers
  - Forward secrecy completely broken
- **Session keys and user data**:
  - Active TLS session keys
  - Usernames, passwords, session cookies
  - Credit card numbers, personal information
- **Memory contents**:
  - Random 64KB chunks of server memory
  - Could contain anything: keys, data, code
  - Repeated requests reveal more memory
- **No evidence of exploitation**: Attack leaves no traces in logs

Heartbleed: The simple attack

Normal heartbeat request:

`Type=1, Length=4, Data="PING"`

Server responds: `"PING"`

Malicious heartbeat request:

`Type=1, Length=65535, Data="A"`

Server responds: `"A" + 65534 bytes of memory`

- **One line of C code**: `memcpy(bp, pl, payload);`
- **Missing check**: `if (payload ≠ 1 + 2 + hblen) error();`
- **Lesson**: Simple bugs can have massive consequences

Heartbleed: Global impact and response

- **Affected systems**:
  - 17% of all SSL/TLS web servers (500,000+ sites)
  - Major services: Yahoo, Flickr, Stack Overflow
  - OpenSSL versions 1.0.1 through 1.0.1f
- **Emergency response**:
  - OpenSSL fixed within days
  - Mass certificate revocation and reissuance
  - Users advised to change all passwords
- **Long-term consequences**:
  - Increased funding for OpenSSL development
  - Core Infrastructure Initiative (CII) formed
  - Better security auditing of critical libraries
- **Branding success**: First security vulnerability with logo and website

SMACK and FREAK: Two attacks from one paper (2015)

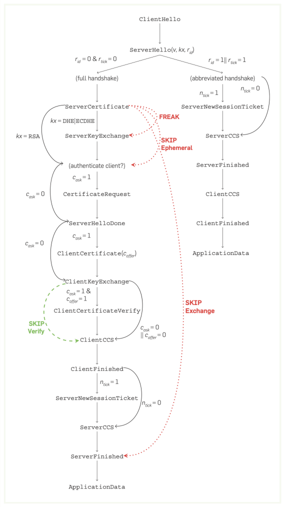
- **Research by**: Inria Prosecco team (again!)[a]
- **Two major attack classes discovered**:
  - **SMACK**: State Machine AttaCKs
  - **FREAK**: Factoring RSA Export Keys
- **SMACK core insight**: TLS implementations have **state machines**
  - Expected message sequence: ClientHello → ServerHello → Certificate → ...
  - Implementations track current state
  - But different implementations have different state machines!
- **FREAK core insight**: Legacy export-grade RSA still supported
  - 512-bit RSA keys can be factored in hours
  - Downgrade attacks force use of weak keys

[a] https://appliedcryptography.page/papers/#smack-tls

SMACK: How state machine attacks work

- **Example scenario**: Web server with two TLS libraries
  - Library A handles initial handshake
  - Library B handles application data
- **Attack technique**:
  1. Send duplicate or out-of-order handshake messages
  2. Library A and Library B enter different states
  3. Library A thinks handshake is complete
  4. Library B thinks handshake is still in progress
- **Result**:
  - Bypass authentication
  - Downgrade security parameters
  - Inject malicious data
- **Discovery method**: Systematic testing with model checking

SMACK: How state machine attacks work



FREAK: Factoring RSA Export Keys

- **Export-grade cryptography legacy**:
  - 1990s US export restrictions required weak crypto
  - 512-bit RSA keys for international software
  - Restrictions lifted, but support remained in implementations
- **FREAK attack flow**:
  1. Client requests strong RSA key exchange
  2. Attacker modifies ClientHello to request export-grade RSA
  3. Server responds with 512-bit RSA parameters
  4. Attacker factors 512-bit RSA key (8-10 hours)
  5. Attacker can decrypt entire TLS session
- **Vulnerable systems**: 36.7% of all browser-trusted sites
- **Impact**: Complete compromise of TLS connections

When math was classified as weapons

- **US Export Administration Regulations (1970s-1990s)**:
  - Cryptographic software classified as "munitions"
  - Same category as tanks, missiles, and fighter jets
  - Export required State Department license (like arms dealing!)
- **The absurd reality**:
  - Mathematical algorithms = weapons of war
  - Publishing crypto code = illegal arms export
  - Explaining RSA algorithm abroad = potential felony

When math was classified as weapons

- **Practical impact**:
  - US software artificially weakened for international markets
  - 40-bit keys for “export grade” crypto (easily breakable)
  - Non-US developers gained competitive advantage
- **The irony**: Trying to keep crypto weak made **everyone** less secure
  - Dual-use problem: same algorithms protect banks and terrorists
  - Weak crypto created systemic vulnerabilities

Bernstein v. United States

- **Daniel J. Bernstein**: Graduate student at UC Berkeley (1990s)
  - Developed “Snuffle” encryption algorithm
  - Wanted to publish academic paper and source code
  - Government: “That’s illegal arms export!”
- **The lawsuit**: Bernstein v. United States (1995-2003)
  - Argued cryptographic code is protected speech
  - First Amendment covers mathematical expressions
  - Government can’t censor academic research

Daniel J. Bernstein

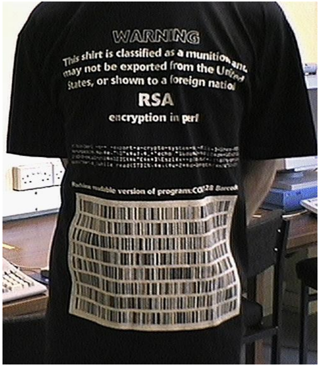
Bernstein v. United States

- **Key victories**:
  - 1996: Court rules source code is speech
  - 1999: 9th Circuit affirms First Amendment protection
  - Export controls on publicly available crypto unconstitutional
- **Legacy**: Opened floodgates for strong cryptography
  - TLS, HTTPS, modern secure communications
  - Academic freedom in cryptographic research
  - Foundation for today's digital security
- **Fun fact**: Bernstein later created Curve25519 (used in TLS 1.3!)

Daniel J. Bernstein

“The Crypto Wars”

RSA tattoo



RSA t-shirt

Netscape “not for export” floppy

The Cypherpunk Manifesto (1993)

- **Key principles**:[a]
  - Cryptography essential for privacy
  - Cannot trust governments/corporations
  - "We must defend our own privacy"
- **"Cypherpunks write code"**:
  - Software defends privacy
  - Code is free for all to use
  - "Software can't be destroyed"
- **Vision**: Cryptography will spread globally, enabling anonymous transactions

[a] https://www.activism.net/cypherpunk/manifesto.html

The Moral Character of Cryptographic Work

Rogaway, 2015

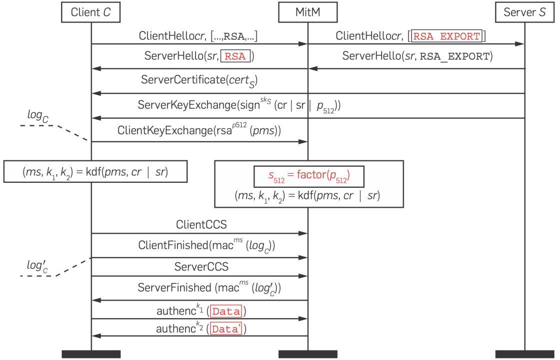
- **Core thesis**: Cryptography is inherently political - it configures power
  - Not just puzzles and math, but tools that shape society
  - Confers intrinsic moral dimension on the field
- **The Snowden wake-up call**:
  - Ordinary people lack basic communication privacy
  - Mass surveillance threatens democracy and human dignity
  - Cryptography's failure: focused on theory, not protecting people
- **Academic cryptography's problems**:
  - Divorced from real-world privacy concerns
  - Serves governments and corporations, not ordinary people
  - Marginalized secure messaging and anti-surveillance work
- **Distinction**: Crypto-for-security (commercial) vs. crypto-for-privacy (social/political)

Rogaway's call to action

- **Cryptographers' moral obligations**:
  - Remember responsibility to humanity
  - Consider societal implications of work
  - Use academic freedom to resist mass surveillance
- **Concrete recommendations**:
  - Develop anti-surveillance technologies
  - Think twice about military funding
  - Work on secure messaging and privacy tools
  - Apply practice-oriented provable security to privacy
- **Vision for the field**:
  - Build cryptographic commons beyond corporate/government reach
  - Make surveillance more expensive
  - Create "boring crypto" that just works for people

FREAK

Figure 4. FREAK attack: a man-in-the-middle downgrades a connection from `RSA` to `RSA_EXPORT`. Then, by factoring the server's 512-bit export-grade RSA key, the attacker can hijack the connection, while the client continues to think it has a secure connection to the server.



SMACK: Real vulnerabilities found

- **miTLS vs. OpenSSL**:
  - Attacker can skip client authentication
  - Exploit differences in certificate validation
- **NSS (Firefox) vulnerabilities**:
  - Early application data acceptance
  - Certificate validation bypass
- **Java JSSE attacks**:
  - Premature transition to application data
  - Authentication bypass in specific configurations
- **The root cause**: Complex state machines without formal verification
- **Solution approach**: Model checking and formal verification
  - Systematically test all possible message sequences
  - Verify implementations match specifications

Logjam: When Diffie-Hellman goes wrong (2015)

- **Research team**: 14 researchers from 10 institutions[a]
- **Target**: Diffie-Hellman key exchange in TLS
- **Two main attacks**:
  - **Logjam**: Downgrade to weak 512-bit DH groups
  - **Precomputation**: Break commonly used 1024-bit groups
- **Context**: 1990s US export restrictions on cryptography
  - "Export-grade" crypto limited to weak parameters
  - Legacy support for 512-bit DH groups remained
- **Fundamental question**: Can we still break today's crypto by exploiting legacy weak parameters?

[a] https://appliedcryptography.page/papers/#imperfect-dh

Logjam: The downgrade attack

- **Attack flow**:
  1. Client offers strong DH groups (2048-bit)
  2. Attacker modifies ClientHello to request weak DH (512-bit)
  3. Server responds with 512-bit DH parameters
  4. Attacker breaks 512-bit DH in real-time
  5. Attacker can now decrypt entire session
- **Breaking 512-bit DH**:
  - Academic cluster: 7 minutes
  - Amazon EC2: under $100 per connection
  - NSA-level resources: near real-time
- **Vulnerable servers**: 8.4% of top 1 million HTTPS sites
- **The irony**: Export restrictions from 1990s still creating vulnerabilities in 2015

Logjam: The downgrade attack

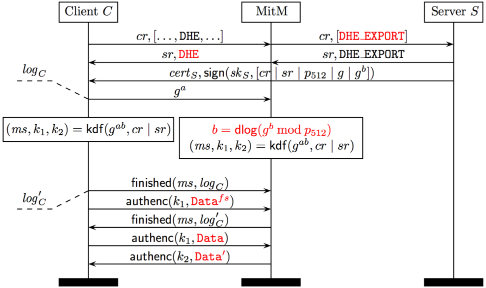


Figure 2: **The Logjam attack.** A man-in-the-middle can force TLS clients to use export-strength DH with any server that allows DHE_EXPORT. Then, by finding the 512-bit discrete log, the attacker can learn the session key and arbitrarily read or modify the contents. $\mathtt{Data}^{fs}$ refers to False Start [30] application data that some TLS clients send before receiving the server's Finished message.

Logjam: Precomputation attacks on 1024-bit groups

- **The number field sieve algorithm**:
  - Most efficient known algorithm for breaking DH
  - Has expensive precomputation phase
  - Once precomputation is done, individual logs are cheaper
- **Attack economics**:
  - Precomputation for 1024-bit group: several months, millions of dollars
  - Individual discrete logs: 30 seconds
  - Amortized cost: profitable for high-value targets
- **Widespread vulnerability**:
  - 18% of top 1M HTTPS sites use single 1024-bit group
  - 66% of VPN servers use same group
  - 26% of SSH servers use same group
- **NSA implications**: Could explain some of NSA's cryptanalytic capabilities

Logjam: Precomputation attacks on 1024-bit groups

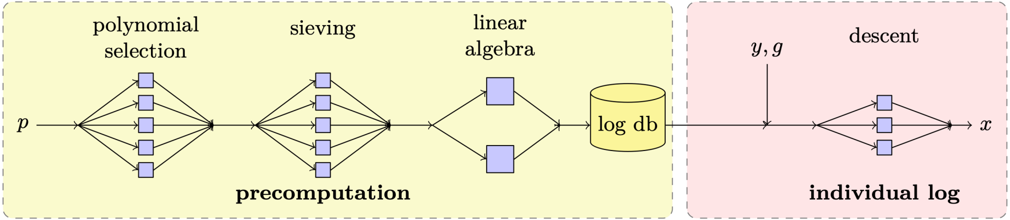


Figure 1: **The number field sieve algorithm for discrete log** consists of a precomputation stage that depends only on the prime p and a descent stage that computes individual logs. With sufficient precomputation, an attacker can quickly break any Diffie-Hellman instances that use a particular p.

Logjam: The “well-known groups” problem

- **Common practice**: Everyone uses the same DH parameters
  - RFC 5114 specifies “standard” groups
  - Apache mod_ssl ships with default parameters
  - Easier than generating custom parameters
- **Concentration risk**:
  - Breaking one group breaks many servers
  - Amortizes the cost of precomputation
  - Creates attractive targets for nation-state actors
- **Timeline for 1024-bit groups**:
  - 2015: Academic resources could break with significant effort
  - 2020: Within reach of well-funded adversaries
  - 2025: Potentially routine for state actors
- **Recommendation**: Move to 2048-bit DH or elliptic curves

Logjam: Intelligence services exploit these issues!

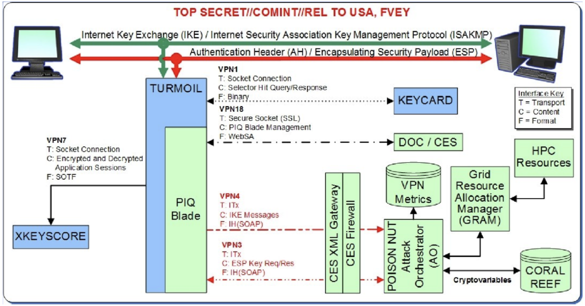


Figure 4: **NSA's VPN decryption infrastructure.** This classified illustration published by Der Spiegel [67] shows captured IKE handshake messages being passed to a high-performance computing system, which returns the symmetric keys for ESP session traffic. The details of this attack are consistent with an efficient break for 1024-bit Diffie-Hellman.

Logjam: Countermeasures and lessons

- **Immediate fixes**:
  - Disable export-grade DH entirely
  - Upgrade to 2048-bit or larger DH groups
  - Prefer elliptic curve Diffie-Hellman (ECDH)
- **Browser responses**:
  - Reject connections with weak DH parameters
  - Implement warnings for short DH keys
- **Broader lessons**:
  - Legacy cryptography creates long-term vulnerabilities
  - Export restrictions had lasting negative security impact
  - Centralized parameters create systemic risks
  - Need to plan for cryptographic algorithm transitions
- **Policy implications**: Demonstrated real-world harm from crypto restrictions

SWEET32: Birthday attacks on 64-bit block ciphers (2016)

- **Researchers**: Karthikeyan Bhargavan and Gaëtan Leurent (Inria)[a]
- **Target**: 64-bit block ciphers (3DES, Blowfish)
- **Core vulnerability**: Birthday paradox in block cipher usage
  - 64-bit blocks → 2^{32} blocks before collisions
  - Long-lived TLS connections can encrypt that much data
  - Collision reveals information about plaintext
- **Attack name**: **SWEET32** - birthday attacks on block ciphers
- **Practical scenario**: HTTPS connections sending repetitive data
  - JavaScript making repeated AJAX requests
  - Cookies or authentication tokens repeated in each request

[a] https://appliedcryptography.page/papers/#inria-sweet32

SWEET32: The birthday attack mechanics

- **Birthday paradox**:
  - For n-bit blocks, expect collision after $2^{n/2}$ blocks
  - 64-bit blocks: collision after $2^{32} = 4$ billion blocks
  - At 1 Mbps: 9 hours, at 10 Mbps: 1 hour
- **Collision exploitation**:
  - When same plaintext block encrypted twice → same ciphertext
  - Attacker identifies when collision occurs
  - Can deduce relationships between plaintext blocks
- **Attack requirements**:
  - Long-lived TLS connection
  - Ability to generate traffic (malicious JavaScript)
  - Repetitive plaintext content (cookies, tokens)
- **Proof of concept**: Extracted HTTP cookies in 30 hours

SWEET32: Real-world impact and remediation

- **Vulnerable systems**:
  - Legacy systems still using 3DES
  - Some VPN implementations
  - Older TLS configurations
- **Attack limitations**:
  - Requires very long connections
  - Needs repetitive plaintext patterns
  - Success rate depends on traffic patterns
- **Countermeasures**:
  - Migrate to 128-bit block ciphers (AES)
  - Implement connection limits (rekeying)
  - Disable 3DES in TLS configurations
- **Browser responses**: Disabled 3DES by default
- **Lesson**: Even "theoretical" attacks can become practical

Transcript Collision Attacks (2016)

- **Researchers**: Karthikeyan Bhargavan and Gaëtan Leurent (Inria)[a]
- **Novel attack class**: Hash collision attacks on protocol transcripts
- **Core idea**:
  - Protocols hash their message transcripts for integrity
  - Find two different transcripts with same hash
  - Substitute one transcript for another
- **Targets**: TLS, IKE (IPsec), SSH
- **Hash collision research**: Building on advances in MD5 and SHA-1
  - Google's SHA-1 collision (SHAttered) published in 2017
  - But collision attacks were becoming practical by 2016

[a] https://appliedcryptography.page/papers/#inria-collisions

Transcript Collision

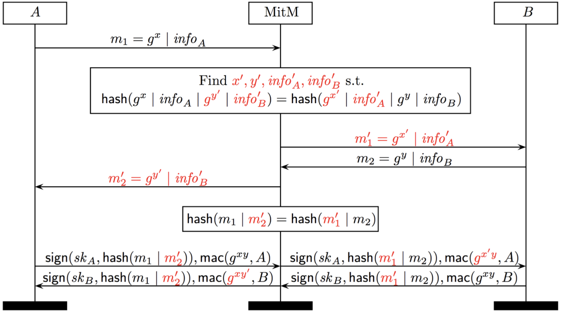


Fig. 2. Man-in-the-middle credential forwarding attack on SIGMA'. The attacker creates a *transcript collision* by tampering with the messages shown in red. At the end of the protocol, the client and server have seemingly authenticated each other, but the attacker knows both connection keys, and hence can read or write any data.

Transcript Collision: Attack on TLS

- **TLS transcript hashing**:
  - TLS computes hash of all handshake messages
  - Used in Finished message for integrity verification
  - Older TLS versions used MD5 and SHA-1
- **Attack scenario**:
  1. Attacker finds two handshake transcripts with same hash
  2. First transcript: legitimate client-server handshake
  3. Second transcript: attacker's malicious handshake
  4. Attacker substitutes malicious transcript
- **Consequences**:
  - Bypass authentication checks
  - Downgrade security parameters
  - Inject malicious content

Transcript Collision: Cross-protocol attacks

- **Cross-protocol confusion**:
  - Same hash function used in multiple protocols
  - IKE and TLS both use SHA-1 for transcript hashing
  - Attacker crafts messages that are valid in both protocols
- **Attack example**:
  - Client connects to TLS server
  - Attacker substitutes IKE handshake with same hash
  - Client's TLS stack processes IKE messages
  - Potential for memory corruption or bypass
- **Mitigation strategies**:
  - Use strong hash functions (SHA-256,SHA-384)
  - Protocol-specific message formats
  - Separate hash contexts for different protocols
- **Lesson**: Hash collisions threaten more than just digital signatures

Qualys SSL Labs

- Free online service to analyze TLS/SSL configuration.
- Tests any public HTTPS server.
- Provides detailed security assessment.
- Available at: `https://www.ssllabs.com/ssltest/`

Section 3

How TLS 1.3 Transformed Protocol Design

TLS 1.3: The great cleanup

- **Problem**: TLS 1.2 inherited decades of legacy features.
  - Suboptimal security and performance.
  - Complex and error-prone configurations.
  - High risk of implementation bugs.
- **Solution**: Complete redesign for TLS 1.3.
  - Kept only the good parts.
  - Added modern security features.
  - Simplified the bloated design.
- **Result**: TLS 1.3 is more secure, efficient, and simpler.
- TLS 1.3 = "mature TLS".

TLS 1.3: Learning from TLS 1.2's mistakes

- **TLS 1.2's legacy problem**:
  - Decades of accumulated features and algorithms
  - Many insecure options still supported for compatibility
  - Complex configurations prone to errors
- **TLS 1.3's philosophy**: **Remove everything dangerous**
  - If it's been broken, remove it
  - If it's complex and error-prone, simplify it
  - If it's not needed, delete it
- **Result**: A much cleaner, more secure protocol

TLS 1.3: Algorithmic spring cleaning

TLS 1.2 supported:

- MD5 (broken)
- SHA-1 (broken)
- RC4 (broken)
- AES-CBC (padding oracles)
- MAC-then-encrypt
- Various weak ciphers

TLS 1.3 only allows:

- Strong hash functions only
- Authenticated encryption
- Modern, secure algorithms
- No legacy cruft

Philosophy: "If you can configure it wrong, remove the option!"

Authenticated encryption: No more MAC-then-encrypt

- **TLS 1.2 approach**: MAC-then-encrypt
  - Compute MAC over plaintext
  - Encrypt (plaintext + MAC)
  - Vulnerable to padding oracle attacks
- **TLS 1.3 approach**: Authenticated encryption only
  - AES-GCM, ChaCha20-Poly1305
  - Encryption and authentication in one step
  - No padding oracle vulnerabilities
- **Benefits**:
  - More efficient (one cryptographic operation)
  - More secure (no composition attacks)
  - Simpler implementation

Supported cryptographic algorithms

- **Authenticated Encryption** (only 3 algorithms):
  - AES-GCM (128-bit or 256-bit keys)
  - AES-CCM (128-bit keys, slightly less efficient than GCM)
  - ChaCha20-Poly1305 (256-bit keys, from RFC 7539)
- **Key Derivation Function (KDF)**:
  - HKDF construction based on HMAC (RFC 5869)
  - Uses SHA-256 or SHA-384 hash functions
- **Diffie-Hellman Key Exchange**:
  - **Elliptic Curves**: 3 NIST curves + Curve25519 + Curve448
  - **Integer Groups**: 2,048, 3,072, 4,096, 6,144, 8,192 bits (RFC 7919)
- **Security note**: 2,048-bit DH provides $<$100-bit security
  - Inconsistent with other 128-bit security choices
  - Still practically impossible to break

TLS 1.3 extensions and variations

- **TLS 1.3 supports many options**:
  - Client certificate authentication
  - Preshared key handshakes
  - Various extensions for specific needs
- **Client certificate authentication**:
  - Server can require client to prove its identity
  - Mutual authentication (both parties verified)
  - Common in enterprise environments
- **Preshared keys (PSK)**:
  - Skip certificate verification
  - Use pre-established shared secrets
  - Faster handshake, but requires prior key distribution
- **Flexibility**: TLS 1.3 adapts to different security requirements.

TLS 1.3: formal verification during the design phase

- **Revolutionary approach**: Protocol design meets formal methods
  - Traditional approach: Design protocol → implement → find bugs → patch
  - TLS 1.3 approach: Design protocol → prove correctness → implement
- **Formal verification**: Mathematical proofs of security properties
  - Prove protocol satisfies security requirements
  - Eliminate entire classes of implementation bugs
  - Higher confidence in security guarantees
- **Industry-academia collaboration**:
  - Inria's Prosecco team (France)
  - Microsoft Research Cambridge (Project Everest)
  - Direct input into IETF standardization process
- **Result**: First cryptographic protocol with machine-checked security proofs *as it was being designed!*

Inria Prosecco: Proving TLS 1.3 security

- **Team Prosecco** (Programming securely with cryptography):
  - Led by Karthikeyan Bhargavan at Inria Paris
  - World-leading experts in cryptographic protocol analysis
- **Key contributions to TLS 1.3**:
  - Formal models of the handshake protocol
  - Machine-checked proofs of key security properties
  - Discovery and prevention of potential attacks
- **Tools and methods**:
  - ProVerif: Automated protocol verification tool
  - Symbolic analysis of cryptographic protocols
  - Found subtle issues before standardization
- **Impact**: Security flaws caught during design, not after deployment

Project Everest: Verified cryptographic implementations

- **Project Everest**: Microsoft Research Cambridge initiative
  - Goal: Provably secure, high-performance cryptographic code
  - From high-level specifications to assembly language
- **F* programming language**:
  - Functional language with dependent types
  - Enables specification and verification of code properties
  - Compiles to efficient C code
- **HACL* cryptographic library**:
  - Verified implementations of crypto primitives
  - ChaCha20, Poly1305, Curve25519, etc.
  - Mathematical proofs of correctness and security
- **miTLS**: Verified TLS implementation
  - Reference implementation with security proofs
  - Demonstrates that formal verification scales to real protocols

ProVerif: Automated protocol verification

- **What is ProVerif?**
  - Automated tool for analyzing cryptographic protocols.
    - `https://proverif.inria.fr`
  - Developed by Bruno Blanchet at Inria Paris.
  - Uses symbolic model of cryptography.
- **How ProVerif works**:
  - Protocol modeled in applied pi-calculus.
  - Automated search for attacks and proofs.
  - Handles unbounded number of protocol sessions.
- **TLS 1.3 verification with ProVerif**:
  - Modeled complete TLS 1.3 handshake protocol.
  - Proved secrecy of session keys.
  - Proved forward secrecy properties.
  - Found potential attacks on early draft versions.

F⋆: Functional programming with verification

- **What is F⋆?**
  - Functional programming language with dependent types.
  - Developed by Microsoft Research and Inria.
  - Enables specification and proof of program properties.
- **F⋆ key features**:
  - Static type system catches bugs at compile-time.
  - Can express and verify complex security properties.
  - Compiles to efficient C code via KreMLin compiler.
- **TLS 1.3 implementation in F⋆**:
  - **miTLS**: Complete TLS 1.3 stack with proofs.
  - **HACL⋆**: Verified crypto library (ChaCha20, Poly1305, Curve25519).
  - **EverCrypt**: Agile crypto provider with algorithm selection.
- **Real-world impact**:
  - HACL⋆ integrated into Firefox, Python, Linux kernel.
  - Proves that verified code can be practical and fast.

Formal verification benefits for TLS 1.3

- **Design-time bug prevention**:
  - Subtle protocol flaws caught before standardization.
  - Avoided costly post-deployment patches.
  - Higher confidence in initial design.
- **Implementation guidance**:
  - Formal specifications guide implementers.
  - Clear mathematical definitions of security properties.
  - Reduced ambiguity in standard documents.
- **Security assurance**:
  - Mathematical proofs complement traditional security analysis.
  - Machine-checked proofs eliminate human error.
  - Covers complex interaction between protocol components.
- **Industry adoption**:
  - HACL* used in Firefox, Linux kernel.
  - Proves formal methods can produce practical code.

Removing dangerous features: The CRIME attack example

- **TLS 1.2 feature**: Optional data compression
  - Reduces bandwidth usage
  - Seemed like a good idea...
- **The CRIME attack**: Compression leaks information
  - Compressed length reveals patterns in plaintext
  - Attackers can inject data and observe compression ratios
  - Can extract secrets like authentication cookies
- **TLS 1.3 solution**: Remove compression entirely
  - No compression = no compression-based attacks
  - Security over efficiency
- **Lesson**: Features that seem helpful can create vulnerabilities

Zero padding: Defeating traffic analysis

- **Traffic analysis attack**:
  - Attackers observe encrypted traffic patterns.
  - Extract info from timing, message sizes, etc.
  - Ciphertext size ≈ plaintext size (reveals message length).
- **TLS 1.3 solution**: Zero padding
  - Add zeros to plaintext before encryption.
  - Inflates ciphertext size.
  - Hides true message length from observers.
- **Example**: 100-byte message + 900 zero bytes = looks like 1000-byte message.

Downgrade protection: Preventing version rollback

- **Downgrade attack scenario**:
  1. Client sends ClientHello supporting TLS 1.3
  2. Attacker modifies message to claim only TLS 1.2 support
  3. Server responds with weaker TLS 1.2 connection
  4. Attacker exploits TLS 1.2 vulnerabilities
- **TLS 1.3 defense**: Magic values in server random
  - Server encodes connection type in first 8 bytes of random value
  - TLS 1.2: `44 4F 57 4E 47 52 44 01`
  - TLS 1.1: `44 4F 57 4E 47 52 44 00`
  - TLS 1.3: Random bytes (no pattern)
- **Attack detection**: Client sees wrong pattern → knows it's under attack!

The magic downgrade detection values

What do those hex values spell?

`44 4F 57 4E 47 52 44 01` = “**DOWNGRD**\x01”
`44 4F 57 4E 47 52 44 00` = “**DOWNGRD**\x00”

- **Clever engineering**: Human-readable sentinel values
- **Easy debugging**: Hex dumps show “DOWNGRD” string
- **Security through visibility**: Makes downgrade attempts obvious
- **Cryptographic protection**: Random value is signed by server
  - Attacker can’t modify it without breaking signature

Performance boost: Single round-trip handshake

TLS 1.2 handshake:

- Client → Server: ClientHello
- Client ← Server: ServerHello + Certificate
- Client → Server: Key exchange
- Client ← Server: Finished
- **2 round trips** before encrypted data

TLS 1.3 handshake:

- Client → Server: ClientHello + Key exchange
- Client ← Server: ServerHello + Certificate + Finished
- **1 round trip** before encrypted data

Performance impact: Hundreds of milliseconds saved per connection

Why single round-trip matters

- **Real-world impact**:
  - High-traffic servers: thousands of connections per second
  - Mobile networks: high latency connections
  - User experience: faster page loads
- **Latency examples**:
  - Local network: 1ms → savings minimal
  - Cross-country: 50ms → saves 50ms per connection
  - Satellite internet: 500ms → saves 500ms per connection!
- **Efficiency gain**: Client sends DH key exchange immediately
  - No waiting for server's algorithm selection
  - Client predicts what server will choose

Session resumption: Even faster connections

- **The idea**: Reuse keys from previous connections
  - Client and server remember shared **preshared key (PSK)**
  - Skip certificate validation in subsequent connections
  - Combine PSK with fresh Diffie-Hellman exchange
- **Security benefits**:
  - Forward secrecy maintained (fresh DH keys)
  - Authentication via MAC instead of certificates
- **Performance benefits**:
  - Faster handshake
  - Less CPU usage (no certificate operations)

0-RTT

- **Zero Round-Trip Time (0-RTT)**:
  - Client sends encrypted data in **first message**
  - No waiting for server response
  - Uses PSK from previous session
- **Process**:
  1. Client: ClientHello + PSK + DH key + **encrypted data**
  2. Server: Processes everything, responds with MAC
  3. Client: Verifies MAC, knows it's talking to right server
- **Performance impact**: Eliminates connection setup delay entirely
- **Use case**: Perfect for frequently-visited websites

0-RTT security considerations

- **Replay attack vulnerability**:
  - Attacker records 0-RTT data packet
  - Replays it to server later
  - Server can't distinguish replay from legitimate connection
    - You can probably formally prove that this problem can't be completely solved without a round trip
- **Mitigation strategies**:
  - Server remembers recent 0-RTT messages
  - Applications design requests to be replay-safe
  - Don't use 0-RTT for sensitive operations
- **Trade-off**: Performance vs. replay protection
  - Perfect for browsing, reading content
  - Dangerous for payments, account changes

TLS 1.3: Summary of improvements

- **Security improvements**:
  - Removed all weak algorithms and dangerous features
  - Authenticated encryption only
  - Downgrade protection
- **Performance improvements**:
  - Single round-trip handshake
  - Session resumption with PSK
  - 0-RTT for repeat connections
- **Simplicity improvements**:
  - Fewer configuration options
  - Standardized elliptic curve formats
  - Cleaner protocol design
- **Result**: More secure, faster, and easier to implement correctly

The future of protocol design

- **TLS 1.3 as a model**:
  - First major protocol with end-to-end formal verification
  - Demonstrates feasibility of formal methods at scale
  - Sets new standard for protocol security assurance
- **Expanding to other protocols**:
  - Signal Protocol (secure messaging)
  - WireGuard VPN protocol
  - Post-quantum cryptographic protocols
- **Challenges ahead**:
  - Scaling formal methods to larger, more complex protocols
  - Training developers in formal verification techniques
  - Balancing mathematical rigor with practical engineering
- **Vision**: All security-critical protocols designed with formal verification
  - Higher security guarantees for everyone
  - Fewer catastrophic vulnerabilities

AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 2: Real-World Cryptography

2.1: Transport Layer Security

Nadim Kobeissi

https://appliedcryptography.page