SSL / TLS Case Study

John Mitchell

Course organization (subject to revision)

- January
  - Two written homeworks: first due next Thursday (9 days)
  - Lectures on case studies (protocols and tools)
  - Choose your project: we'll start giving examples Thursday
- February
  - Project presentation #1: describe your system (5-10 min)
  - Lectures on additional approaches
  - Project presentation #2: describe security properties
- March
  - Project presentation #3: results of study

Grading: 20% homework, 30% project presentations, 50% project results and final presentation

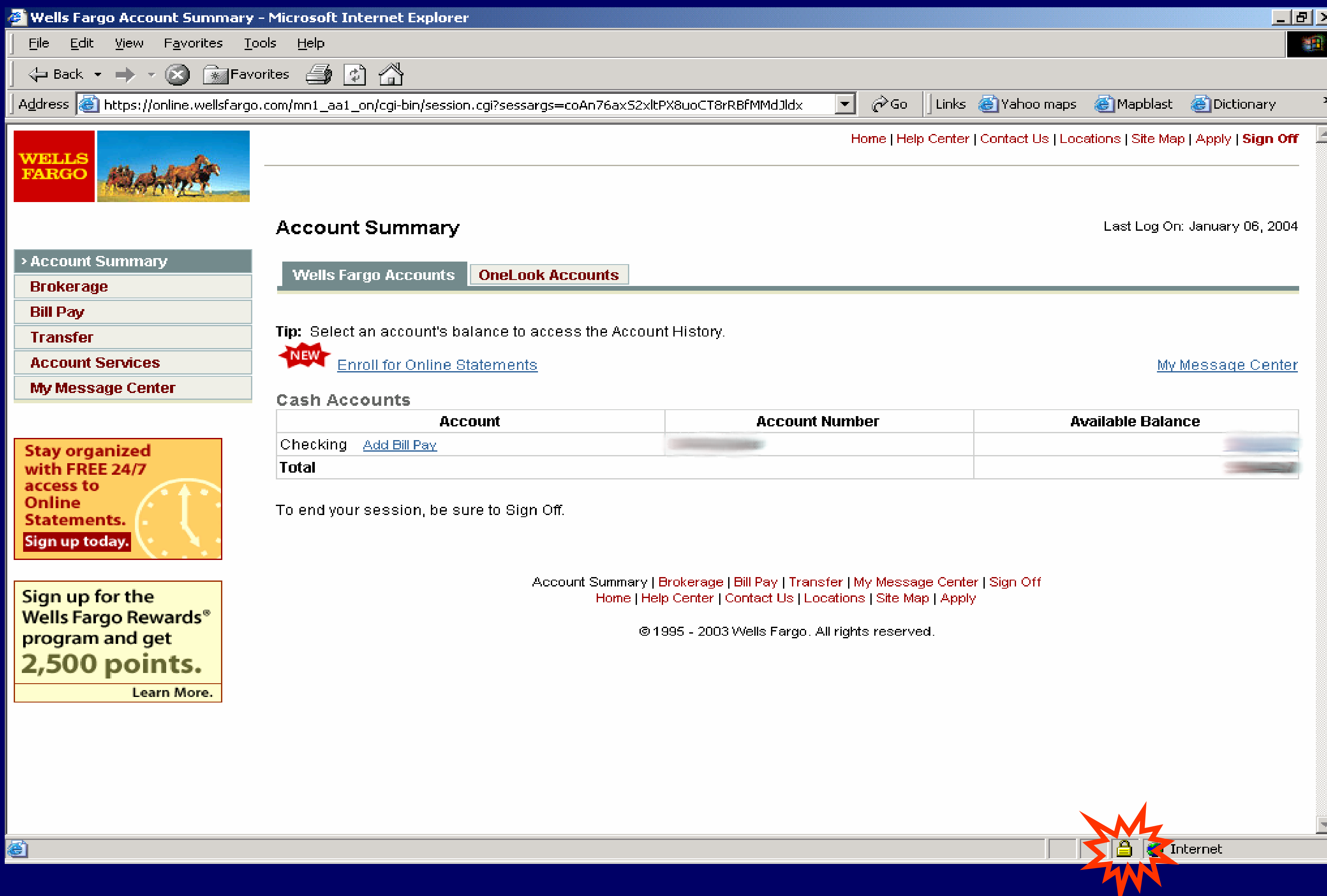
Overview

- Introduction to the SSL / TLS protocol
  - Widely deployed, "real-world" security protocol
- Protocol analysis case study
  - Start with the RFC describing the protocol
  - Create an abstract model and code it up in Murφ
  - Specify security properties
  - Run Murφ to check whether security properties are satisfied
- This lecture is a compressed version of what you would do if SSL were your project!

What is SSL / TLS?

- Transport Layer Security protocol, ver 1.0
  - De facto standard for Internet security
  - “The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications”
  - In practice, used to protect information transmitted between browsers and Web servers
- Based on Secure Sockets Layers protocol, ver 3.0
  - Same protocol design, different algorithms
- Deployed in nearly every web browser

SSL / TLS in the Real World



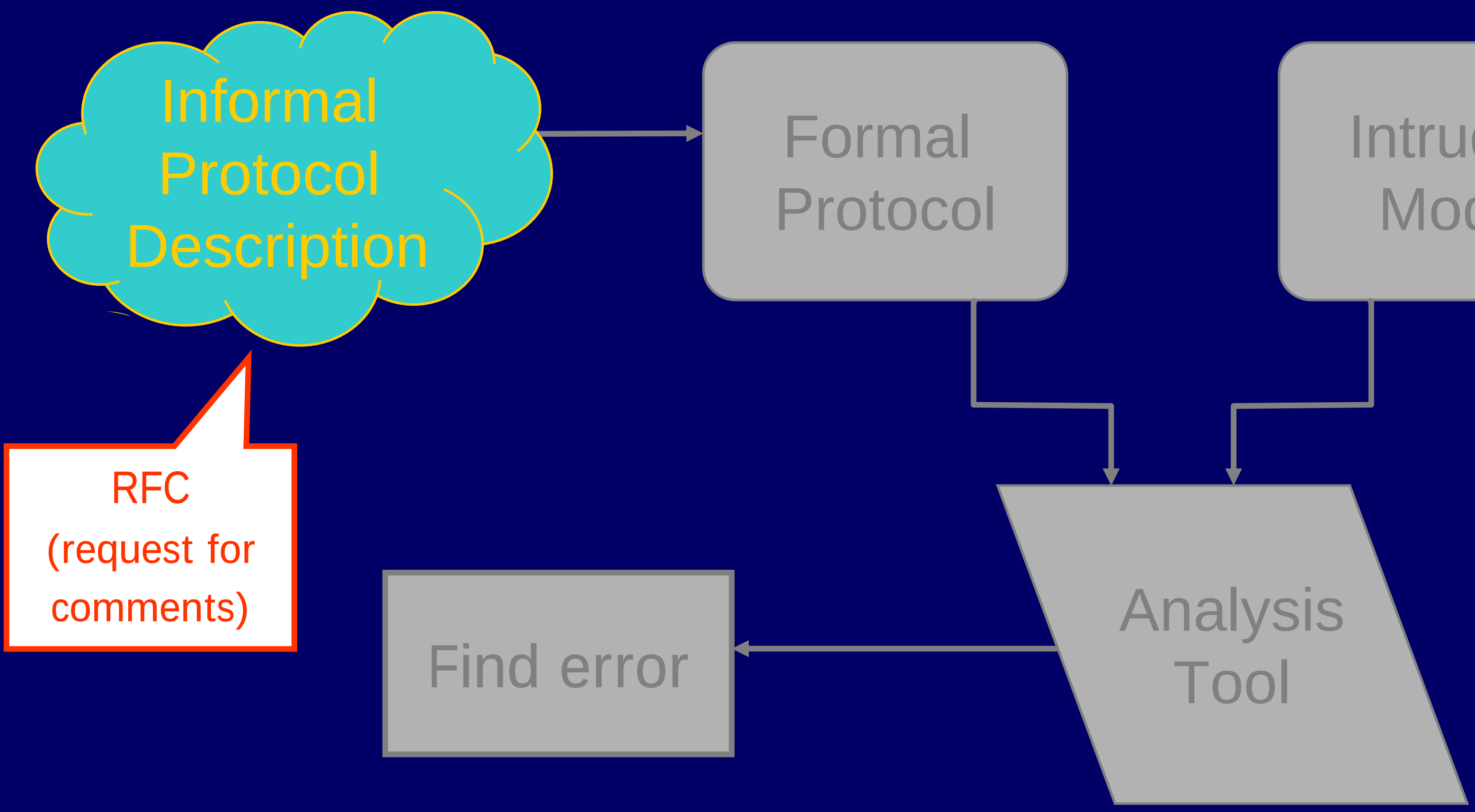
History of the Protocol

- SSL 1.0
  - Internal Netscape design, early 1994?
  - Lost in the mists of time
- SSL 2.0
  - Published by Netscape, November 1994
  - Several problems (next slide)
- SSL 3.0
  - Designed by Netscape and Paul Kocher, November 1996
- TLS 1.0
  - Internet standard based on SSL 3.0, January 1999
  - <u>Not</u> interoperable with SSL 3.0

SSL 2.0 Vulnerabilities

- Short key length
  - In export-weakened modes, SSL 2.0 unnecessarily weakens the authentication keys to 40 bits.
- Weak MAC construction
- Message integrity vulnerability
  - SSL 2.0 feeds padding bytes into the MAC in block cipher modes, but leaves the padding-length unauthenticated, may allow active attackers to delete bytes from the end of messages
- Ciphersuite rollback attack
  - An active attacker may edits the list of ciphersuite preferences in the hello messages to invisibly force both endpoints to use a weaker form of encryption
  - “Least common denominator" security under active attack

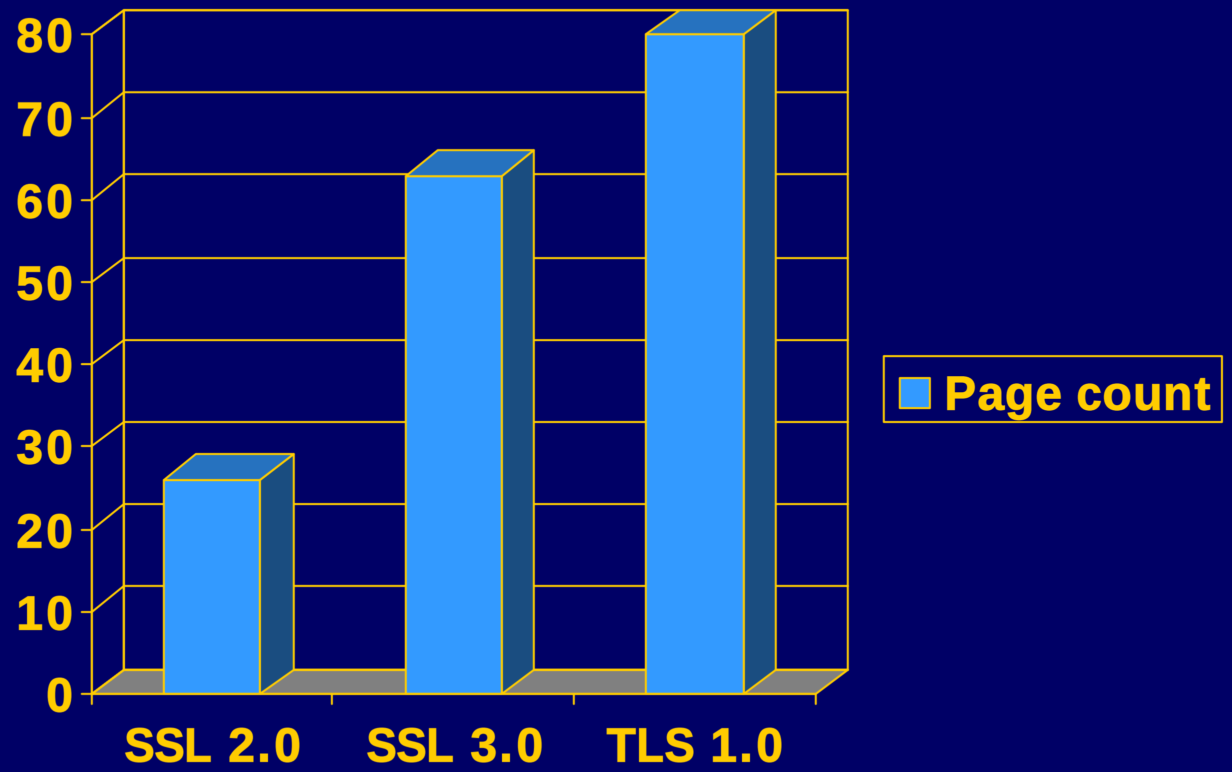
Let’s get going with SSL/TLS ...



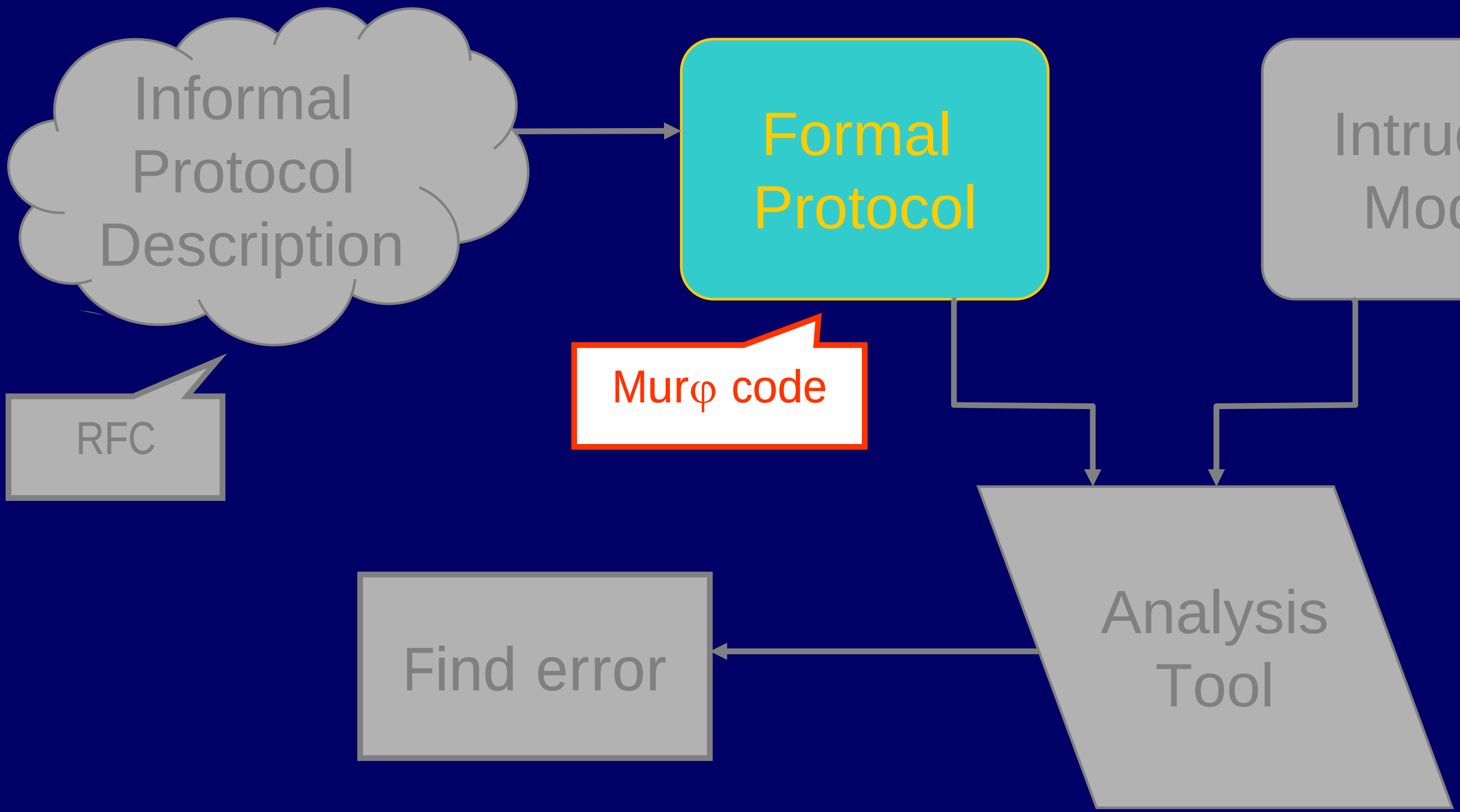
Request for Comments

- Network protocols are defined in an RFC
- TLS version 1.0 is described in RFC 2246
- Intended to be a self-contained definition of the protocol
  - Describes the protocol in sufficient detail for readers who will be implementing it and those who will be doing protocol analysis (that's you!)
  - Mixture of informal prose and pseudo-code
- Read some RFCs to get a flavor of what protocols look like when they emerge from the committee

Evolution of the SSL/TLS RFC



From RFC to Murφ Model



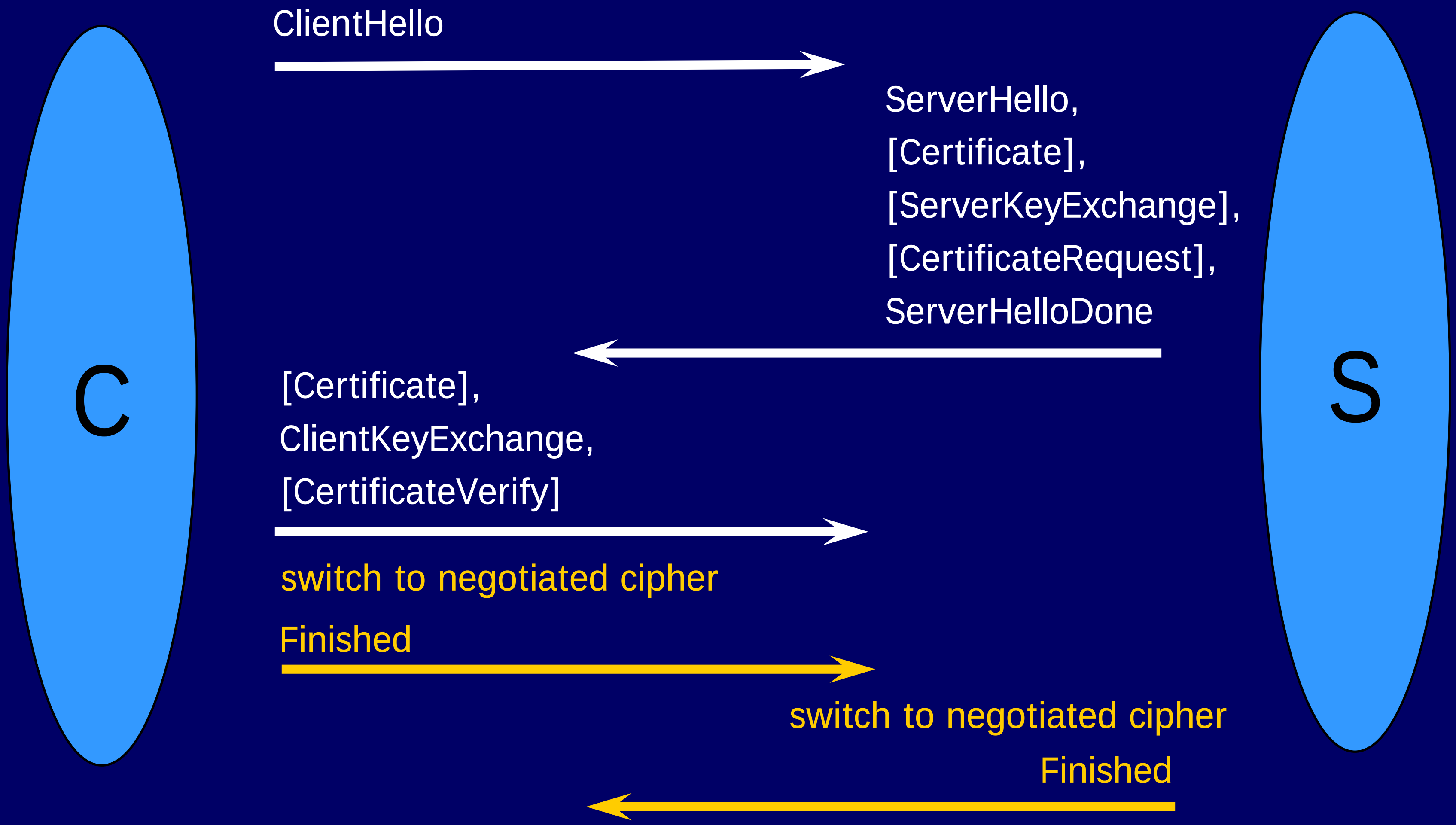
TLS Basics

- TLS consists of two protocols
- Handshake protocol
  - Use public-key cryptography to establish a shared secret key between the client and the server
- Record protocol
  - Use the secret key established in the handshake protocol to protect communication between the client and the server
- We will focus on the handshake protocol

TLS Handshake Protocol

- Two parties: client and server
- Negotiate version of the protocol and the set of cryptographic algorithms to be used
  - Interoperability between different implementations of the protocol
- Authenticate client and server (optional)
  - Use digital certificates to learn each other’s public keys and verify each other’s identity
- Use public keys to establish a shared secret

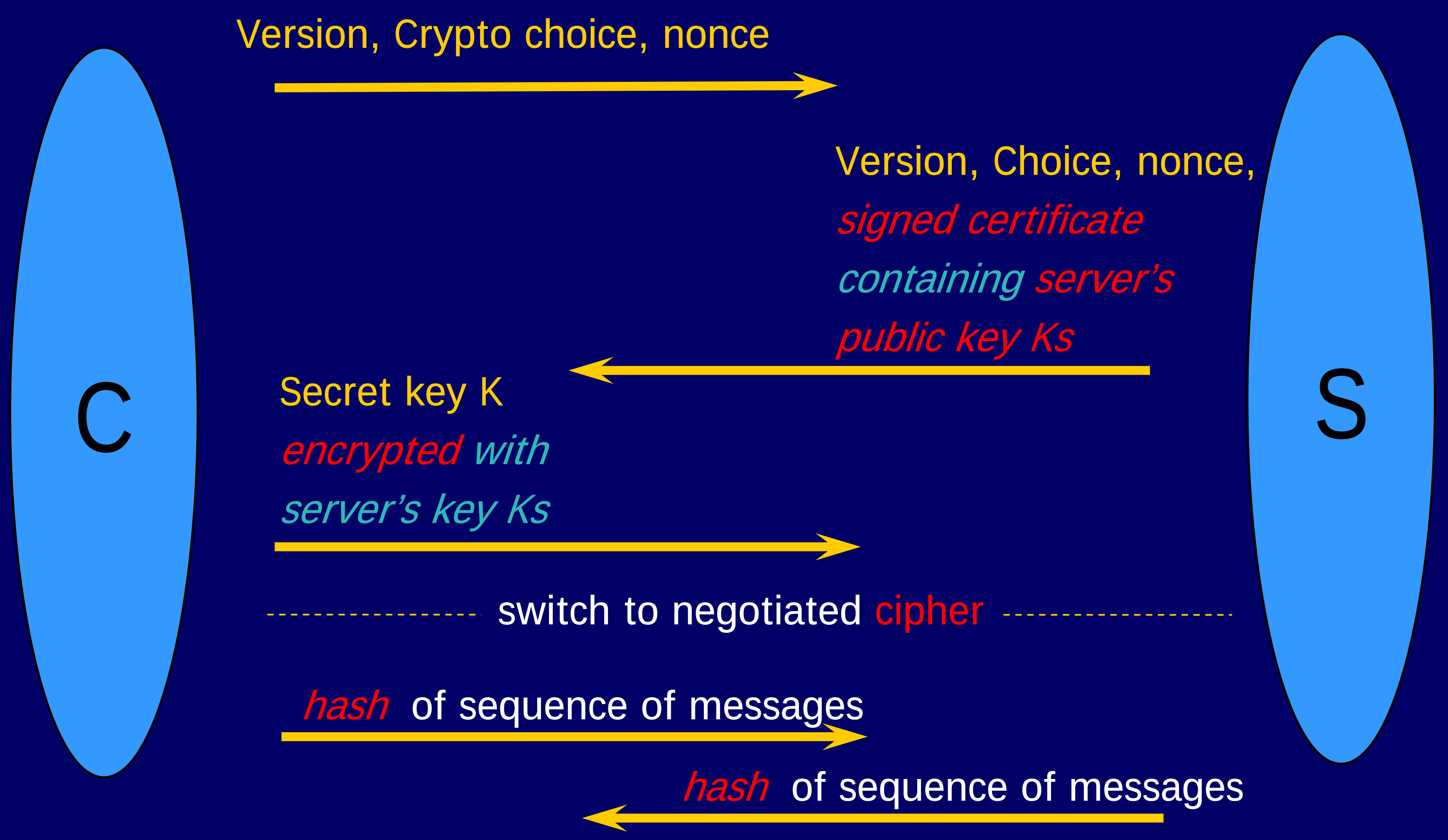
Handshake Protocol Structure



Recall: Basic Cryptographic Concepts

- Encryption scheme
  - functions to encrypt, decrypt data
  - key generation algorithm
- Secret key vs. public key
  - Public key: publishing *key* does not reveal key^{-1}
  - Secret key: more efficient, generally *key* = key^{-1}
- Hash function, MAC
  - Map input to short hash; ideally, no collisions
  - MAC (keyed hash) used for message integrity
- Signature scheme
  - Functions to sign data, verify signature

Use of cryptography



SSL/TLS Cryptography Summary

- Public-key encryption
  - Key chosen secretly (handshake protocol)
  - Key material sent encrypted with public key
- Symmetric encryption
  - Shared (secret) key encryption of data packets
- Signature-based authentication
  - Client can check signed server certificate
  - And vice-versa, in principal
- Hash for integrity
  - Client, server check hash of sequence of messages
  - MAC used in data packets (record protocol)

Public-Key Infrastructure

Known public signature verification key Ka

Server certificate can be verified by any client that has CA verification key Ka

Certificate authority is “off line”

Another general idea in SSL

- Client, server communicate

- Compare hash of all messages
  - Compute hash(hi,hello,howareyou?) locally
  - Exchange hash values under encryption
- Abort if intervention detected

SSL/TLS in more detail ...

ClientHello $C \to S$ $C, Ver_C, Suite_C, N_C$

ServerHello $S \to C$ $Ver_S, Suite_S, N_S, sign_{CA}\{ S, K_S \}$

ClientVerify $C \to S$ $sign_{CA}\{ C, V_C \}$

$\{ Ver_C, Secret_C \}_{K_S}$

$sign_C \{ Hash(Master(N_C, N_S, Secret_C) + Pad_2 +$

$Hash(Msgs + C + Master(N_C, N_S, Secret_C) + Pad_1)) \}$

-------- Change to negotiated cipher --

ServerFinished $S \to C$ $\{ Hash(Master(N_C, N_S, Secret_C) + Pad_2 +$

$Hash(Msgs + S + Master(N_C, N_S, Secret_C) + Pad_1))$

$\}_{Master(N_C, N_S, Secret_C)}$

ClientFinished $C \to S$ $\{ Hash(Master(N_C, N_S, Secret_C) + Pad_2 +$

$Hash(Msgs + C + Master(N_C, N_S, Secret_C) + Pad_1))$

$\}_{Master(N_C, N_S, Secret_C)}$

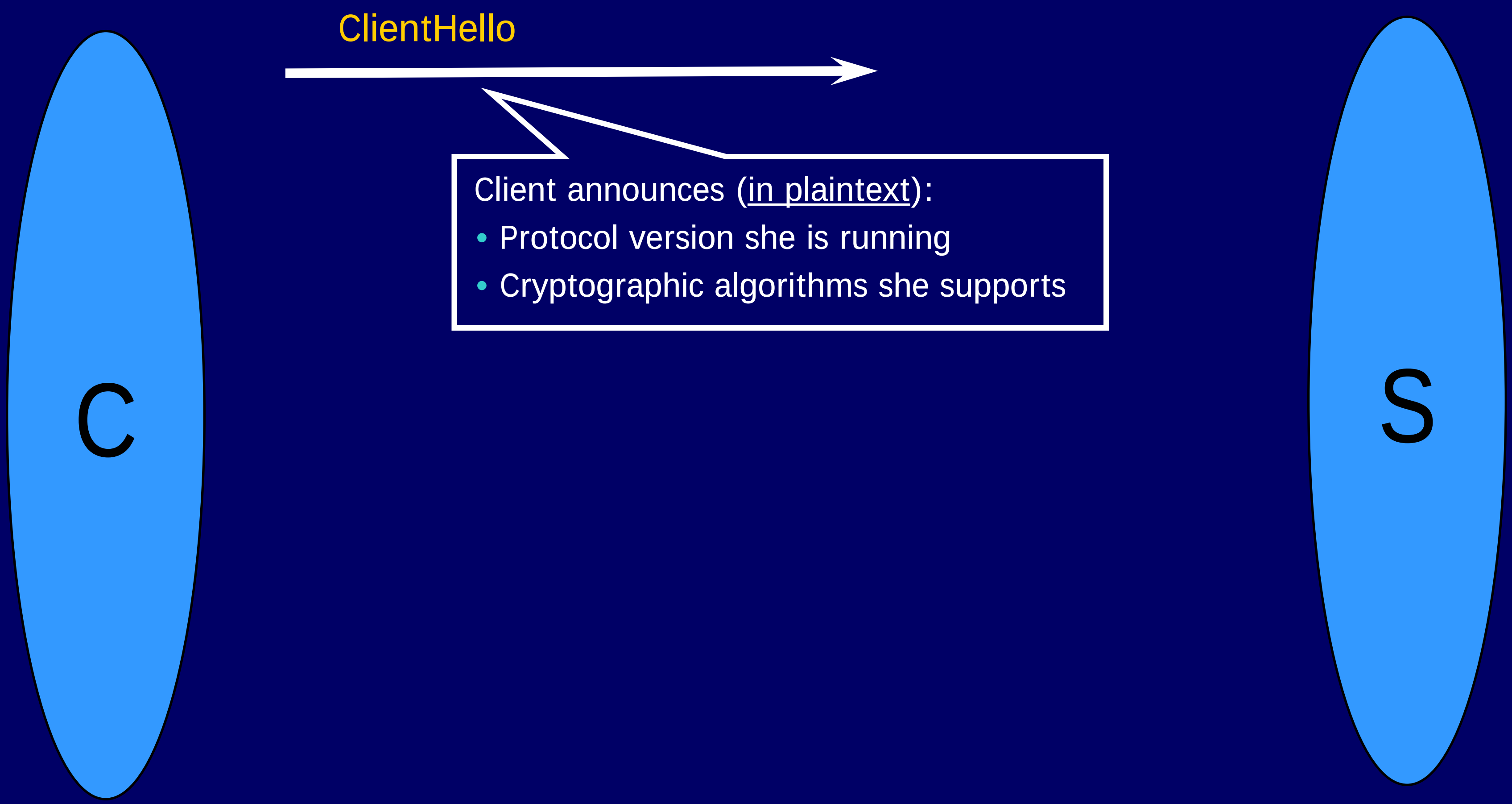
Abbreviated Handshake

- The handshake protocol may be executed in an abbreviated form to resume a previously established session
  - No authentication, key material not exchanged
  - Session resumed from an old state
- For complete analysis, have to model both full and abbreviated handshake protocol
  - This is a common situation: many protocols have several branches, subprotocols for error handling, etc.

Rational Reconstruction

- Begin with simple, intuitive protocol
  - Ignore client authentication
  - Ignore verification messages at the end of the handshake protocol
  - Model only essential parts of messages (e.g., ignore padding)
- Execute the model checker and find a bug
- Add a piece of TLS to fix the bug and repeat
  - Better understand the design of the protocol

Protocol Step by Step: ClientHello



ClientHello (RFC)

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello
```

Highest version of the protocol supported by the client

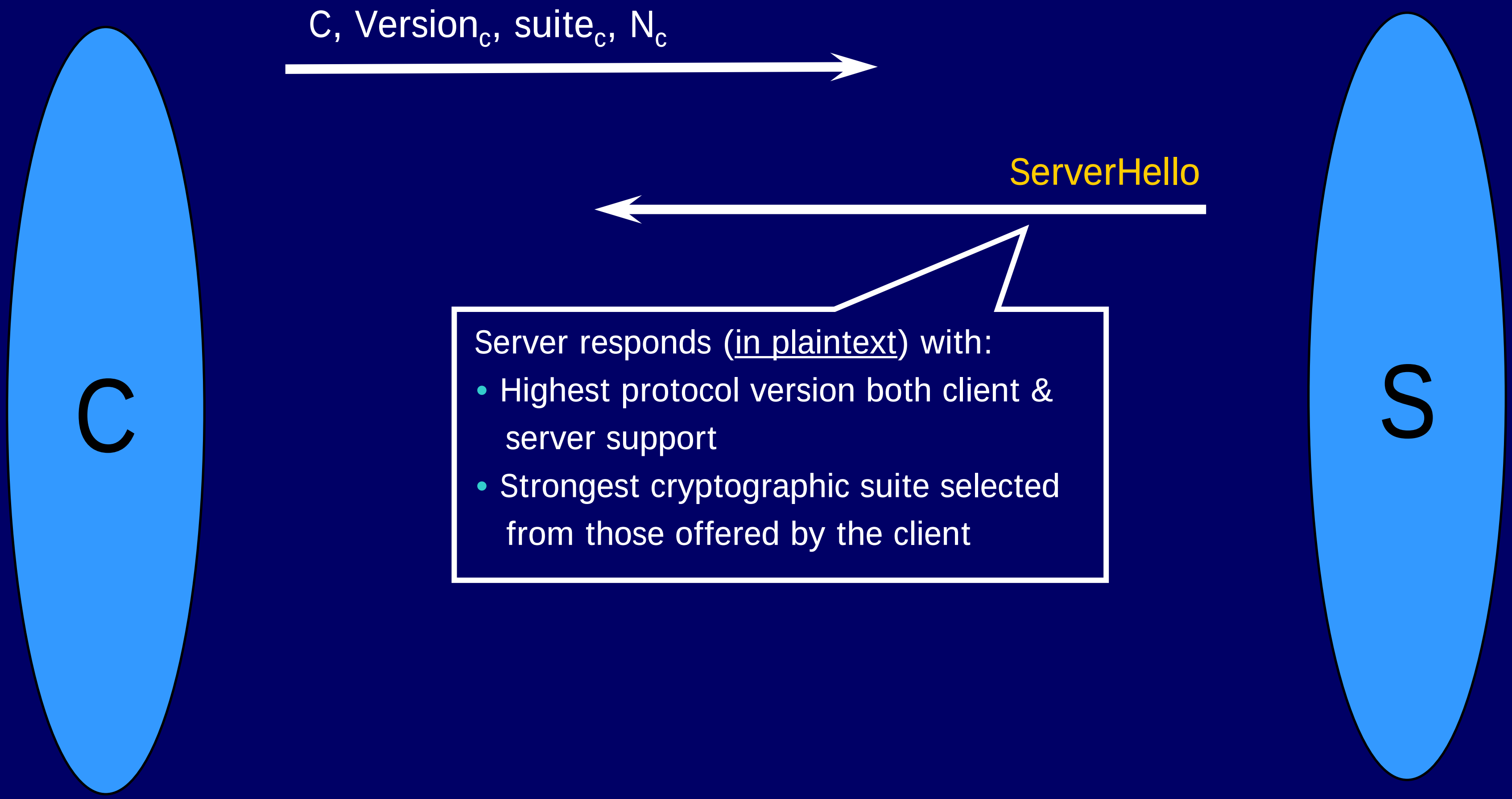
Session id (if the client wants to resume an old session)

Cryptographic algorithms supported by the client (e.g., RSA or Diffie-Hellman)

ClientHello (Murφ)

```
ruleset i: ClientId do
  ruleset j: ServerId do
    rule "Client sends ClientHello to server (new session)"
      cli[i].state = M_SLEEP &
      cli[i].resumeSession  = false
    ==>
    var
      outM: Message;  -- outgoing message
    begin
      outM.source := i;
      outM.dest := j;
      outM.session := 0;
      outM.mType := M_CLIENT_HELLO;
      outM.version := cli[i].version;
      outM.suite := cli[i].suite;
      outM.random := freshNonce();
      multisetadd (outM, cliNet);
      cli[i].state  := M_SERVER_HELLO;
end;  end;  end;
```

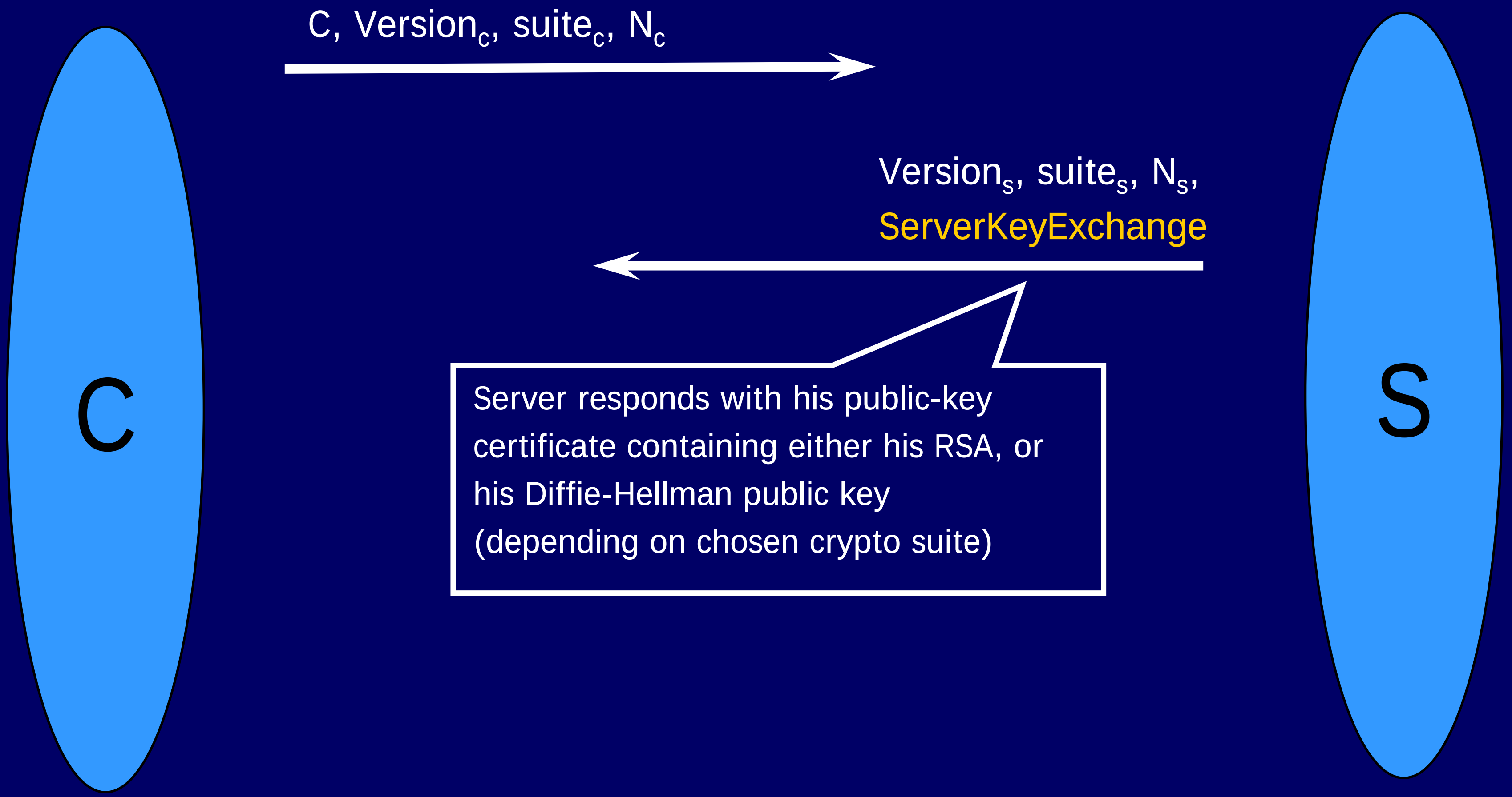
ServerHello



ServerHello (Murφ)

```
ruleset i: ServerId do
  choose l: serNet do
   rule "Server receives ServerHello (new session)"
     ser[i].clients[0].state = M_CLIENT_HELLO &
     serNet[l].dest = i &
     serNet[l].session = 0
    ==>
    var
      inM:   Message;   -- incoming message
      outM: Message;   -- outgoing message
    begin
      inM := serNet[l];  -- receive message
      if inM.mType = M_CLIENT_HELLO then
         outM.source := i;
         outM.dest := inM.source;
         outM.session := freshSessionId();
         outM.mType := M_SERVER_HELLO;
         outM.version := ser[i].version;
         outM.suite := ser[i].suite;
         outM.random := freshNonce();
         multisetadd (outM, serNet);
         ser[i].state  := M_SERVER_SEND_KEY;
    end;  end;  end;
```

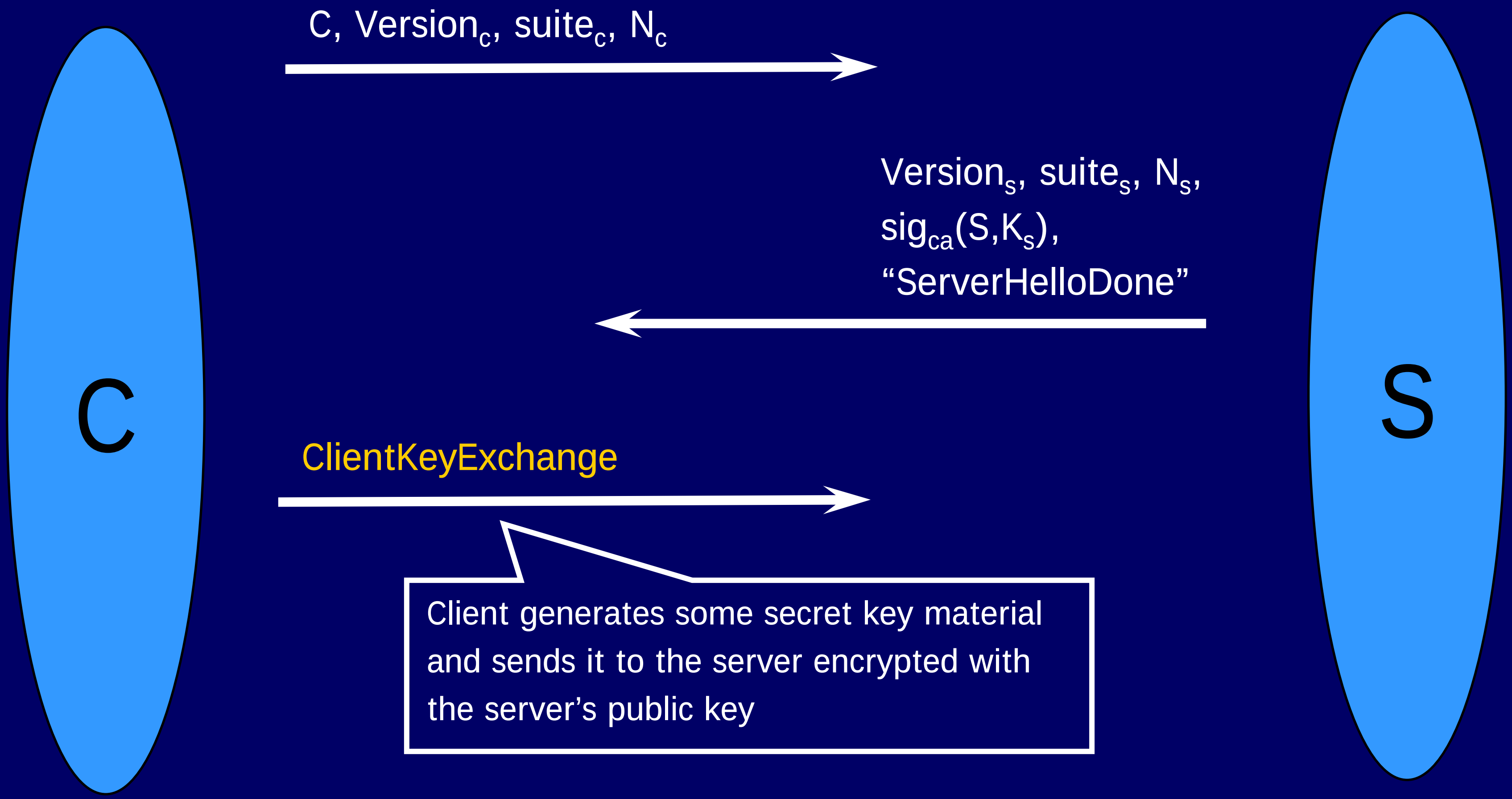
ServerKeyExchange



“Abstract” Cryptography

- We will use abstract data types to model cryptographic operations
  - Assumes that cryptography is perfect
  - No details of the actual cryptographic schemes
  - Ignores bit length of keys, random numbers, etc.
- Simple notation for encryption, signatures, hashes
  - $\{M\}_k$ is message M encrypted with key k
  - $sig_k(M)$ is message M digitally signed with key k
  - hash(M) for the result of hashing message M with a cryptographically strong hash function

ClientKeyExchange

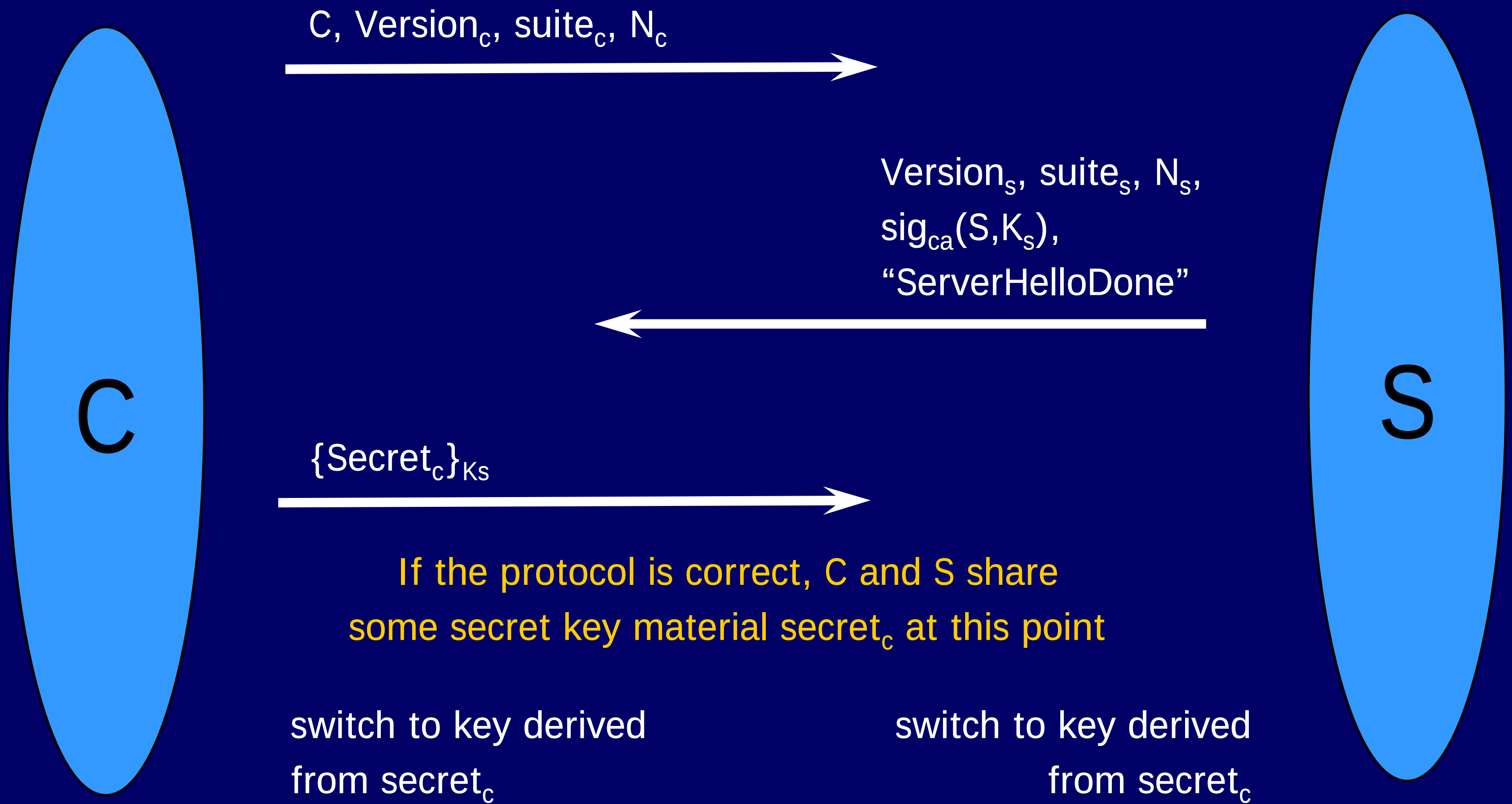


ClientKeyExchange (RFC)

Let's model this as $\{Secret_c\}_{Ks}$

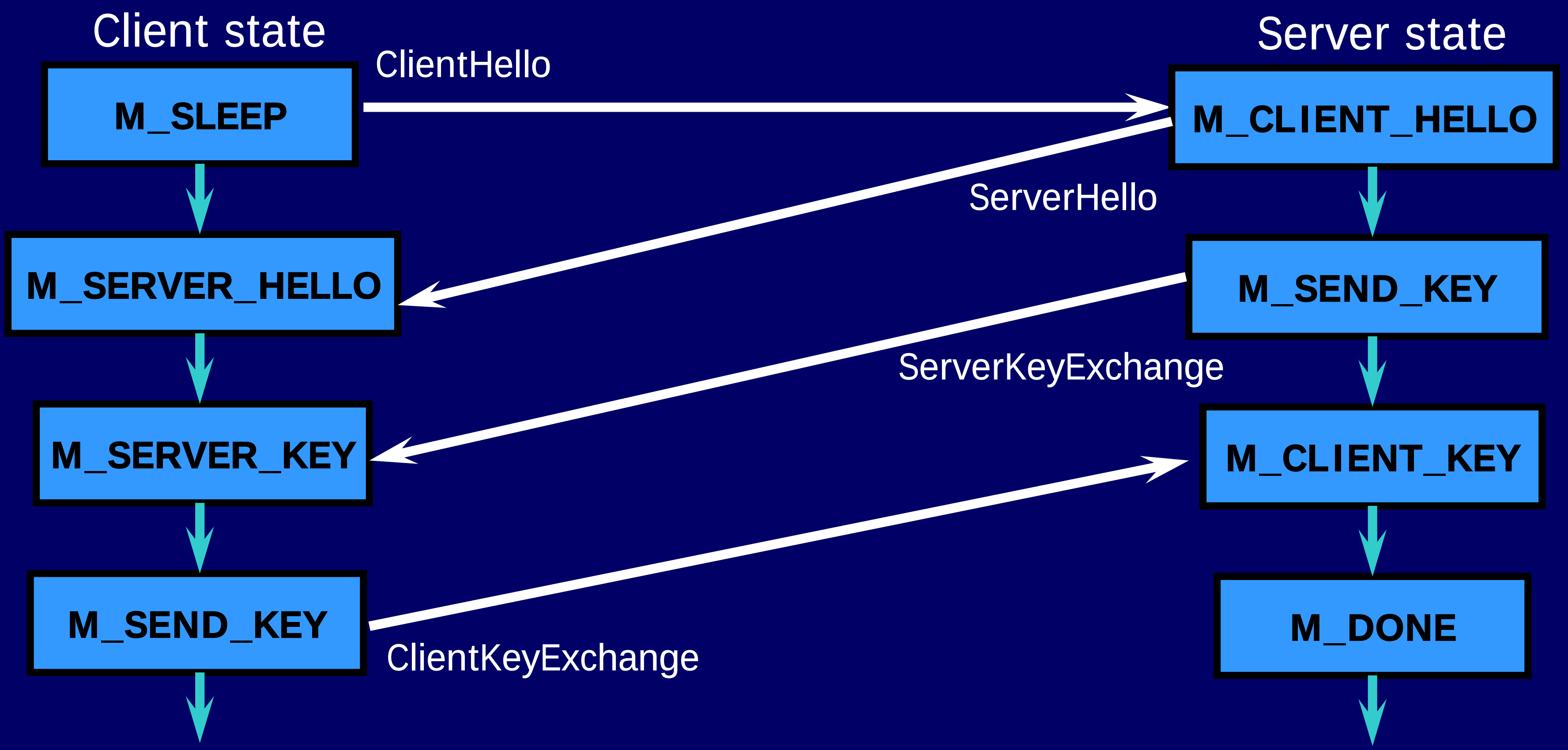
```
struct {
  select (KeyExchangeAlgorithm) {
    case rsa: EncryptedPreMasterSecret;
    case diffie_hellman: ClientDiffieHellmanPublic;
  } exchange_keys
} ClientKeyExchange

struct {
  ProtocolVersion client_version;
  opaque random[46];
} PreMasterSecret
```

“Core” SSL

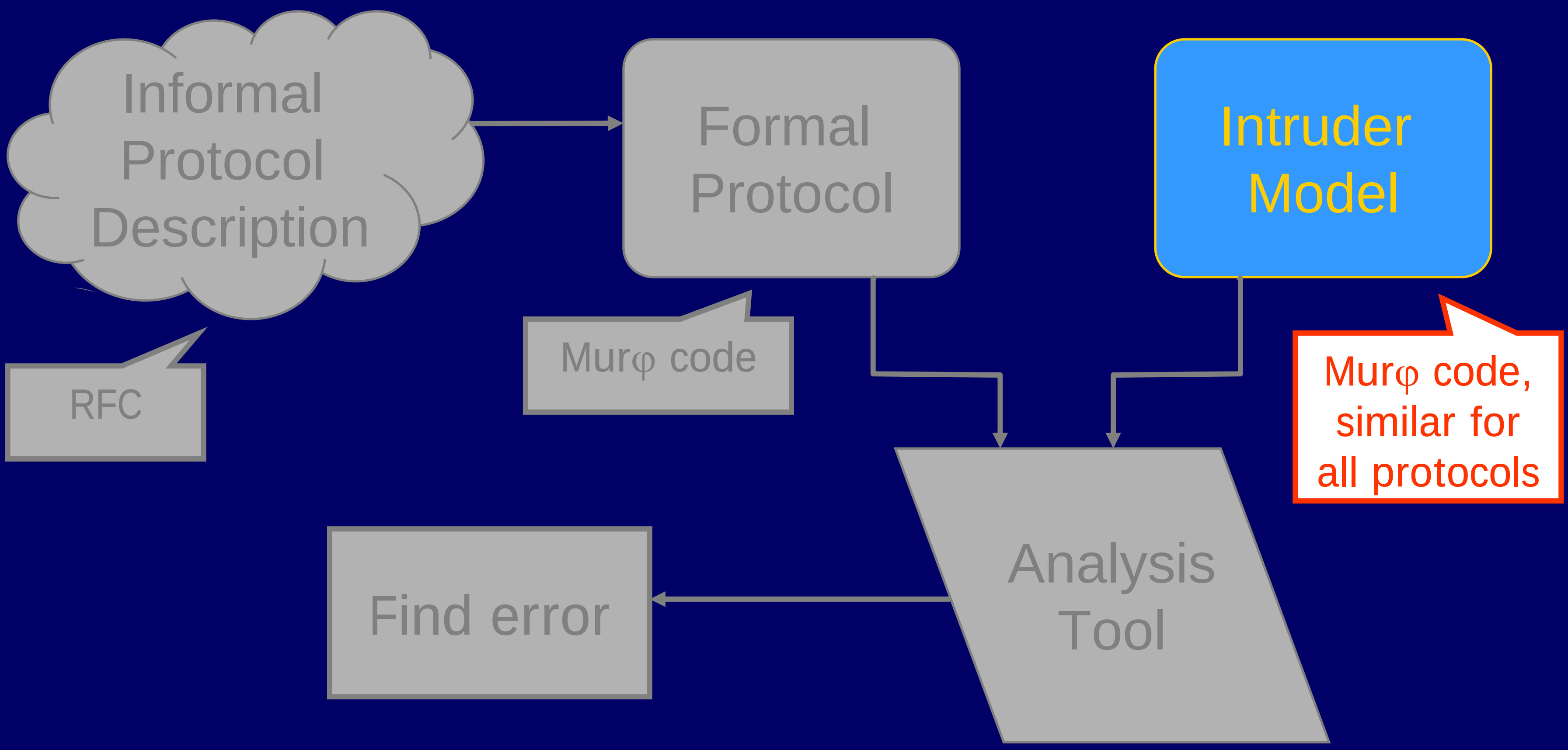

C, Version_c, suite_c, N_c
Version_s, suite_s, N_s,
$\text{sig}_{ca}(S,K_s)$,
“ServerHelloDone”
$\{\text{Secret}_c\}_{Ks}$
If the protocol is correct, C and S share
some secret key material secret_c at this point
switch to key derived from secret_c
switch to key derived from secret_c
C
S

Participants as Finite-State Machines

Murφ rules define a finite-state machine for each protocol participant



Intruder Model



Intruder Can Intercept

- Store a message from the network in the data structure modeling intruder's "knowledge"

```
ruleset i: IntruderId do
  choose l: cliNet do
    rule "Intruder intercepts client's message"
      cliNet[l].fromIntruder = false
    ==>
    begin
      alias msg: cliNet[l] do   -- message from the net
      ...
      alias known: int[i].messages do
         if multisetcount(m: known,
                          msgEqual(known[m], msg)) = 0 then
            multisetadd(msg, known);
         end;
      end;
    end;
```

Intruder Can Decrypt if Knows Key

- If the key is stored in the data structure modeling intruder's "knowledge", then read message

```
ruleset i: IntruderId do
  choose l: cliNet do
    rule "Intruder intercepts client's message"
      cliNet[l].fromIntruder = false
    ==>
    begin
      alias msg: cliNet[l] do   -- message from the net
      ...
      if msg.mType = M_CLIENT_KEY_EXCHANGE then
           if keyEqual(msg.encKey, int[i].publicKey.key) then
              alias sKeys: int[i].secretKeys do
                if multisetcount(s: sKeys,
                   keyEqual(sKeys[s], msg.secretKey)) = 0 then
                   multisetadd(msg.secretKey, sKeys);
                end;
           end;
      end;
```

Intruder Can Create New Messages

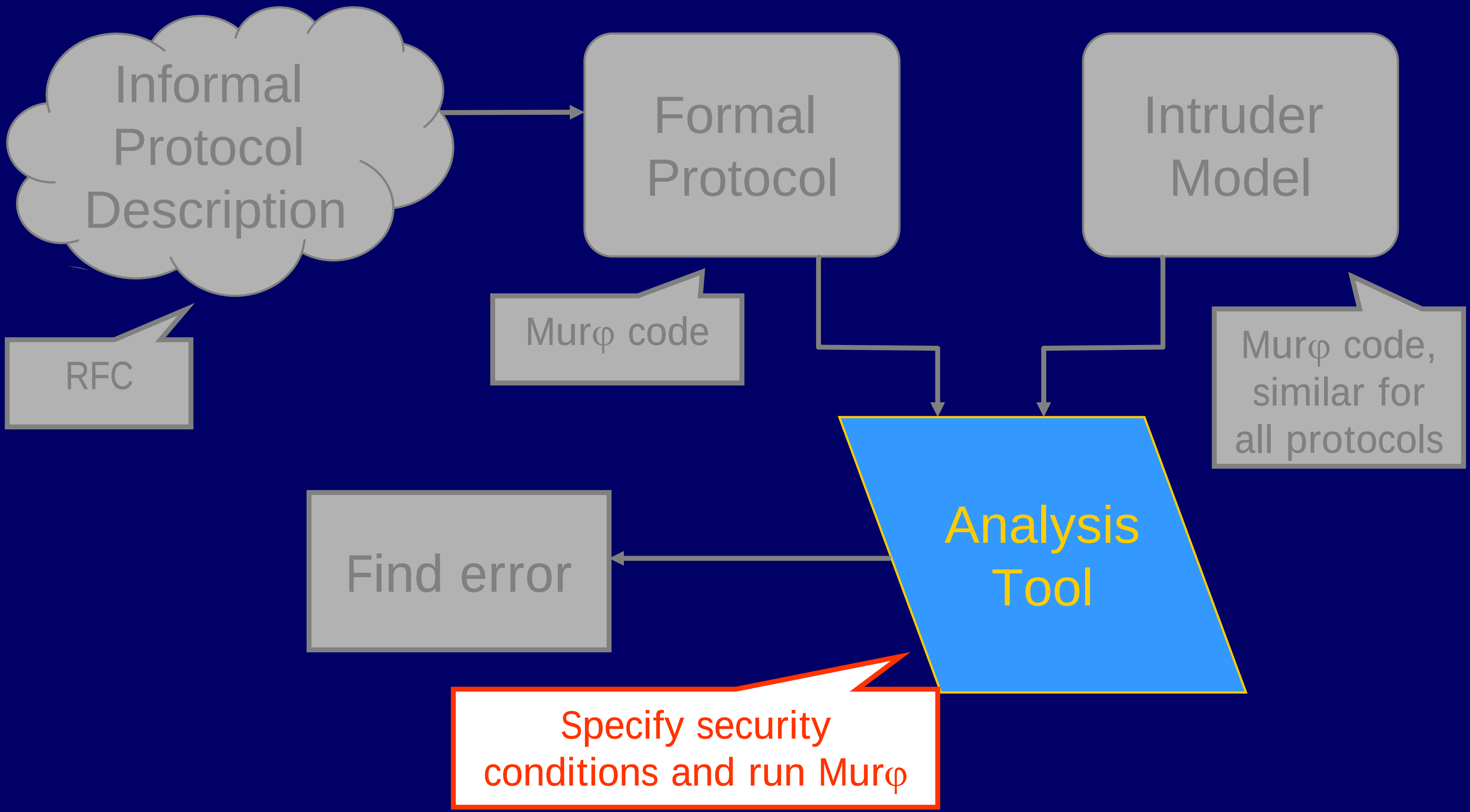
- Assemble pieces stored in the intruder’s “knowledge” to form a message of the right format

```
ruleset i: IntruderId do
  ruleset d: ClientId do
    ruleset s: ValidSessionId do
      choose n: int[i].nonces do
      ruleset version: Versions do
      rule "Intruder generates fake ServerHello"
          cli[d].state = M_SERVER_HELLO
        ==>
        var
          outM: Message;  -- outgoing message
        begin
          outM.source := i; outM.dest := d; outM.session := s;
          outM.mType := M_SERVER_HELLO;
          outM.version := version;
          outM.random := int[i].nonces[n];
          multisetadd (outM, cliNet);
        end; end; end; end;
```

Intruder Model and Cryptography

- There is no actual cryptography in our model
  - Messages are marked as “encrypted” or “signed”, and the intruder rules respect these markers
- Our assumption that cryptography is perfect is reflected in the absence of certain intruder rules
  - There is no rule for creating a digital signature with a key that is not known to the intruder
  - There is no rule for reading the contents of a message which is marked as “encrypted” with a certain key, when this key is not known to the intruder
  - There is no rule for reading the contents of a “hashed” message

Running Murφ Analysis



Secrecy

- Intruder should not be able to learn the secret generated by the client

```
ruleset i: ClientId do
  ruleset j: IntruderId do
    rule "Intruder has learned a client's secret"
      cli[i].state = M_DONE &
      multisetcount(s: int[j].secretKeys,
         keyEqual(int[j].secretKeys[s], cli[i].secretKey)) > 0
    ==>
    begin
      error "Intruder has learned a client's secret"
    end;
  end;
end;
```

Shared Secret Consistency

- After the protocol has finished, client and server should agree on their shared secret

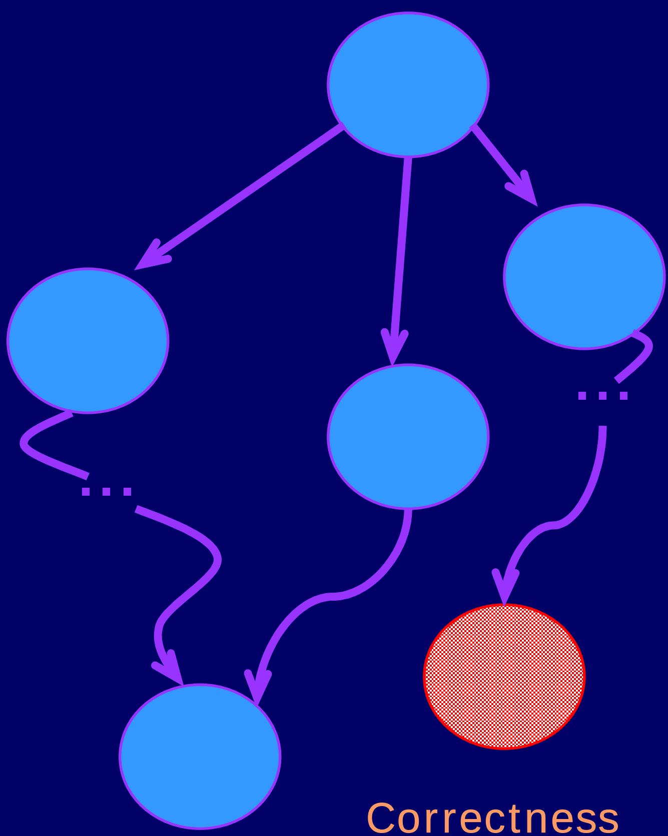
```
ruleset i: ServerId do
  ruleset s: SessionId do
    rule "Server's shared secret is not the same as its client's"
      ismember(ser[i].clients[s].client, ClientId) &
      ser[i].clients[s].state = M_DONE &
      cli[ser[i].clients[s].client].state = M_DONE &
      !keyEqual(cli[ser[i].clients[s].client].secretKey,
                ser[i].clients[s].secretKey)
    ==>
    begin
      error "S's secret is not the same as C's"
    end;
  end;
end;
```

Version and Crypto Suite Consistency

- Client and server should be running the highest version of the protocol they both support

```
ruleset i: ServerId do
  ruleset s: SessionId do
    rule "Server has not learned the client's version or suite correctly"
      !ismember(ser[i].clients[s].client, IntruderId) &
      ser[i].clients[s].state = M_DONE &
      cli[ser[i].clients[s].client].state = M_DONE &
      (ser[i].clients[s].clientVersion != MaxVersion |
       ser[i].clients[s].clientSuite.text != 0)
    ==>
    begin
      error "Server has not learned the client's version or suite correctly"
    end;
  end;
end;
```

Finite-State Verification



Correctness condition violated

- Murφ rules for protocol participants and the intruder define a nondeterministic state transition graph
- Murφ will exhaustively enumerate all graph nodes
- Murφ will verify whether specified security conditions hold in every reachable node
- If not, the path to the violating node will describe the attack

When Does Murφ Find a Violation?

- Bad abstraction
  - Removed too much detail from the protocol when constructing the abstract model
  - Add the piece that fixes the bug and repeat
  - This is part of the rational reconstruction process
- Genuine attack
  - Yay! Hooray!
  - Attacks found by formal analysis are usually quite strong: independent of specific cryptographic schemes, OS implementation, etc.
  - Test an implementation of the protocol, if available

“Basic” SSL 3.0

C

C, $Version_c$=3.0, $suite_c$

$Version_s$=3.0, $suite_s$
$sig_{ca}(S,K_s)$,
“ServerHelloDone”

$\{Secret_c\}_{Ks}$

S

If the protocol is correct, C and S share some secret key material $secret_c$ at this point

switch to key derived from $secret_c$

switch to key derived from $secret_c$

Version Consistency Fails!

C, Version_c=**2.0**, suite_c

Server is fooled into thinking he is communicating with a client who supports only SSL 2.0

Version_s=**2.0**, suite_s

$\text{sig}_{ca}(S,K_s)$,

"ServerHelloDone"

C

S

$\{\text{Secret}_c\}_{Ks}$

C and S end up communicating using SSL 2.0

(weaker earlier version of the protocol)

A Case of Bad Abstraction

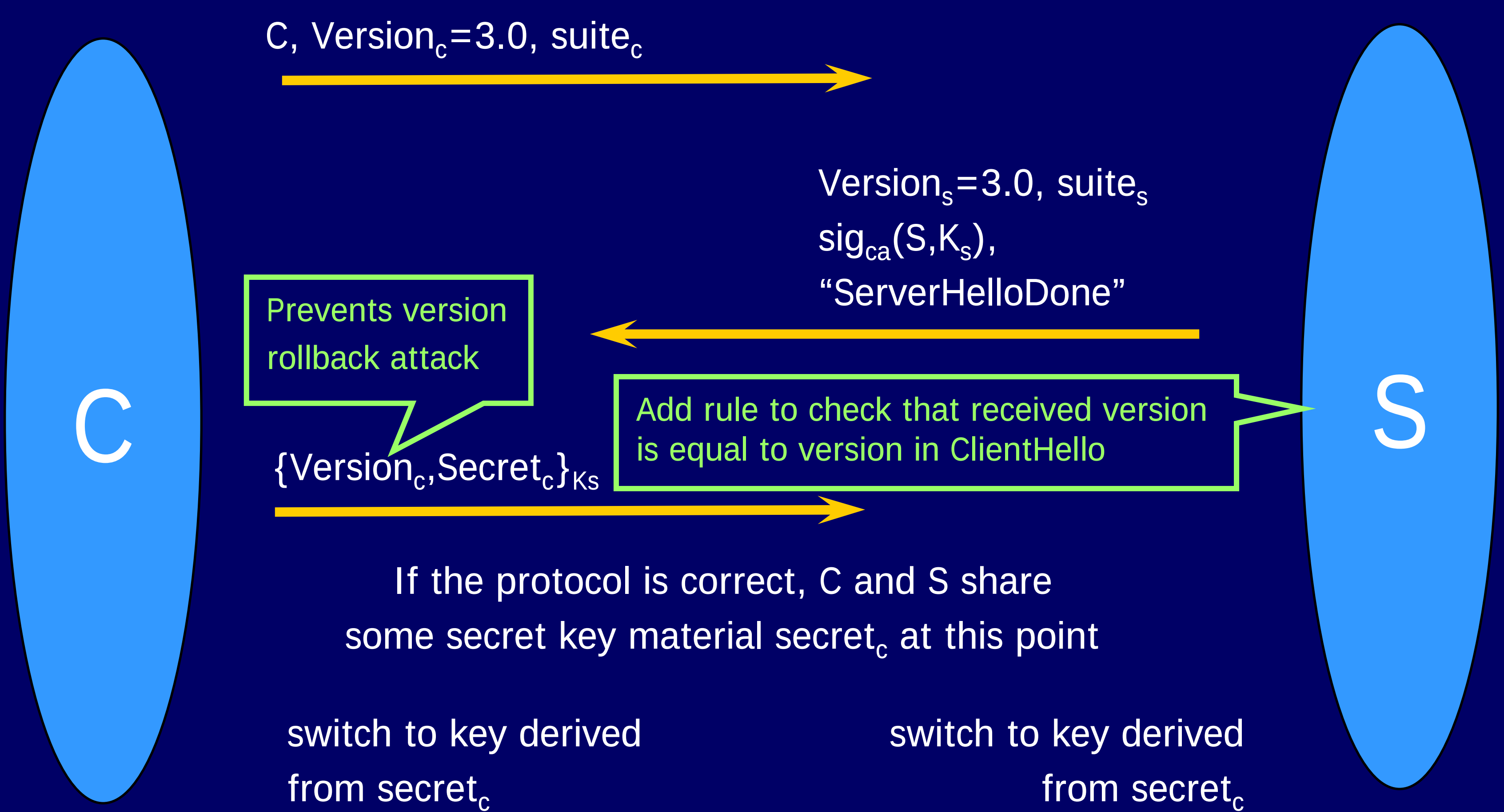
Model this as $\{Version_c, Secret_c\}_{Ks}$

```
struct {
  select (KeyExchangeAlgorithm) {
    case rsa: EncryptedPreMasterSecret;
    case diffie_hellman: ClientDiffieHellmanPublic;
  } exchange_keys
} ClientKeyExchange
```

This piece matters! Need to add it to the model.

```
struct {
  ProtocolVersion client_version;
  opaque random[46];
} PreMasterSecret
```

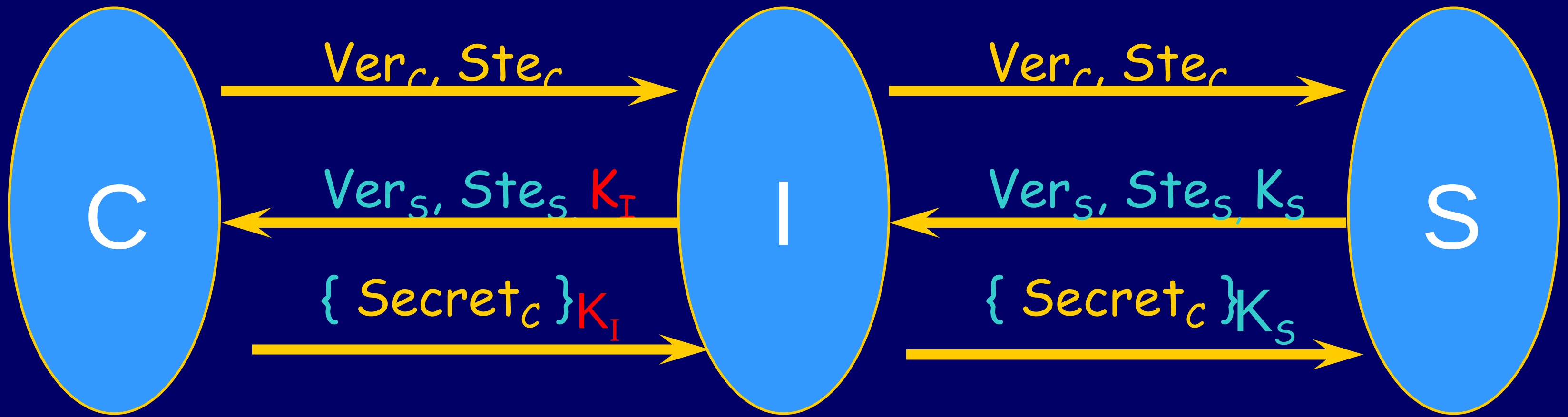
Better “basic” SSL



Summary of Incremental Protocols

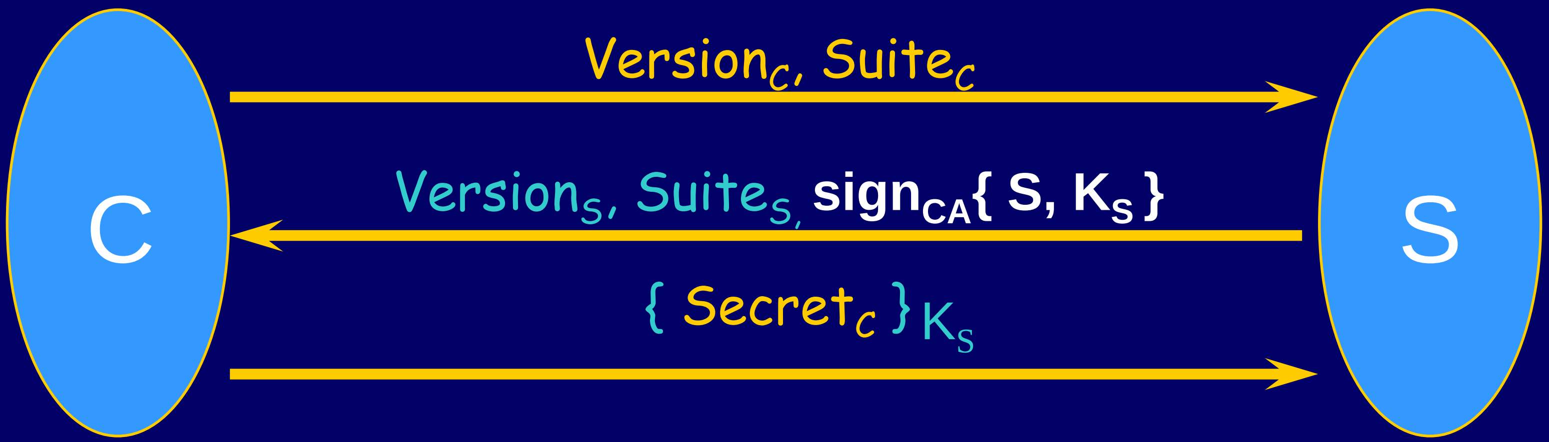
- A = Basic protocol
- B = A + version consistency check
- D = B + certificates for both public keys
  - Authentication for client + Authentication for server
- E = D + verification (Finished) messages
  - Prevention of version and crypto suite attacks
- F = E + nonces
  - Prevention of replay attacks
- G = “Correct” subset of SSL
  - Additional crypto considerations (black art) give SSL 3.0

Attack on Protocol B



- Intruder in the middle
  - Replaces server key by intruder’s key
  - Intercepts secret from client
  - Simulates client to server, server to client

Solution: Certificate Authority

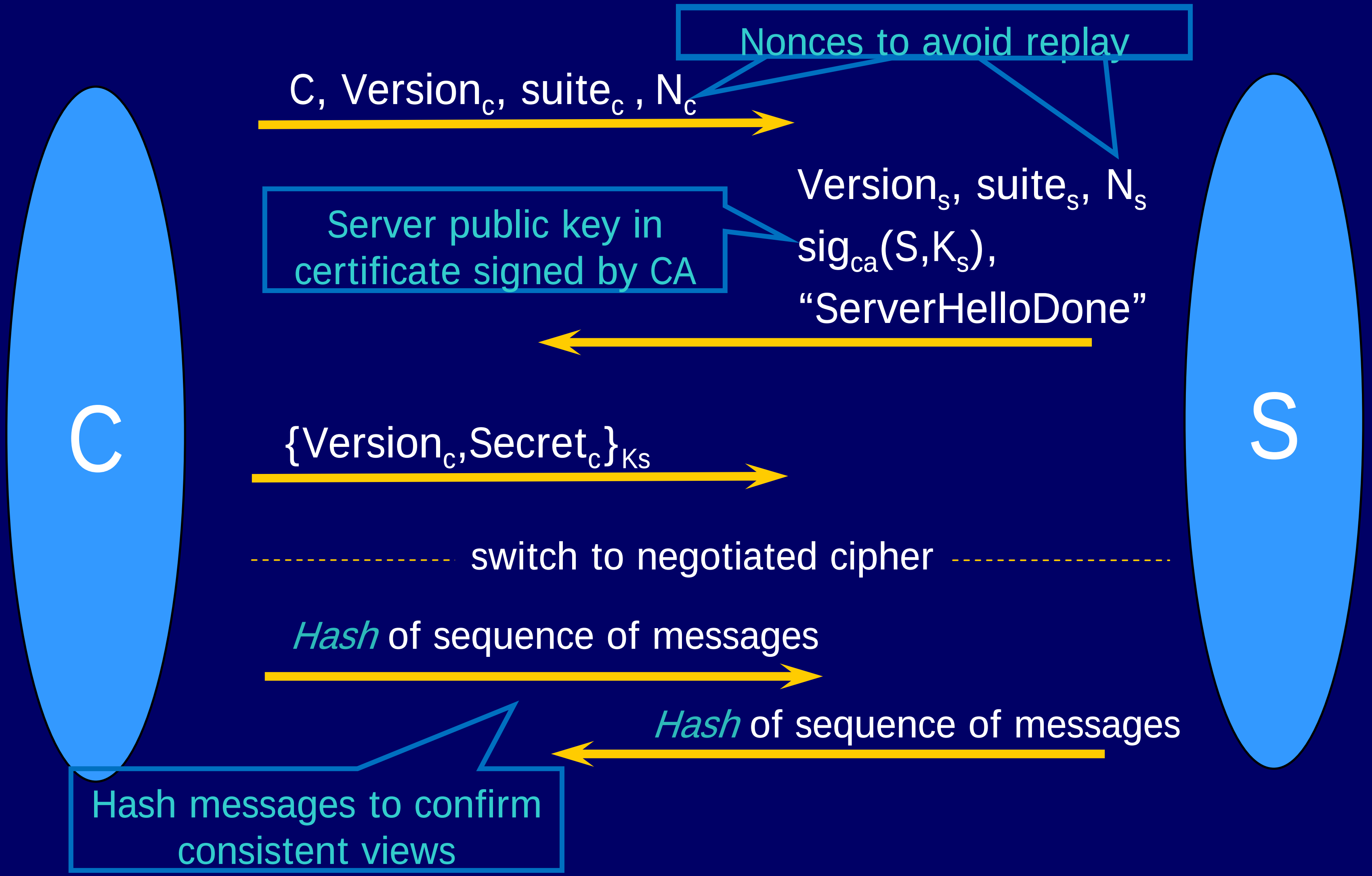


- Defeats previous attack
  - But client is not authenticated to the server …

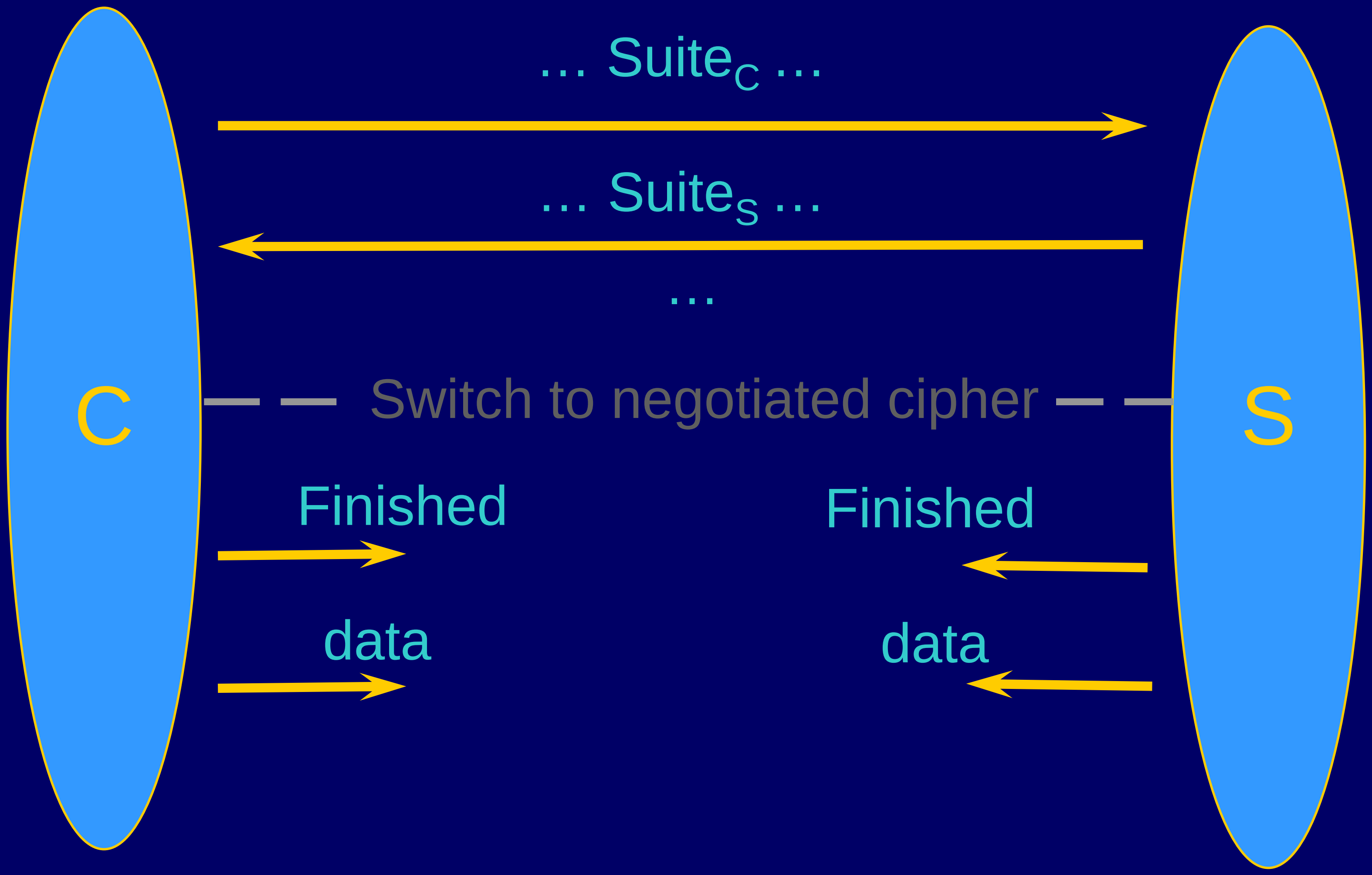
Replay Attacks

- Network eavesdropper can record messages
- If protocol is deterministic, then
  - Eavesdropper can replay client messages to server, OR
  - Eavesdropper can replay server message to client
- This is a problem
  - In each session, each party should be guaranteed that the other is a live participant in the session
- Solution
  - Each run of each protocol should contain at least one new value generated by each party, included in messages, and checked before session is considered done

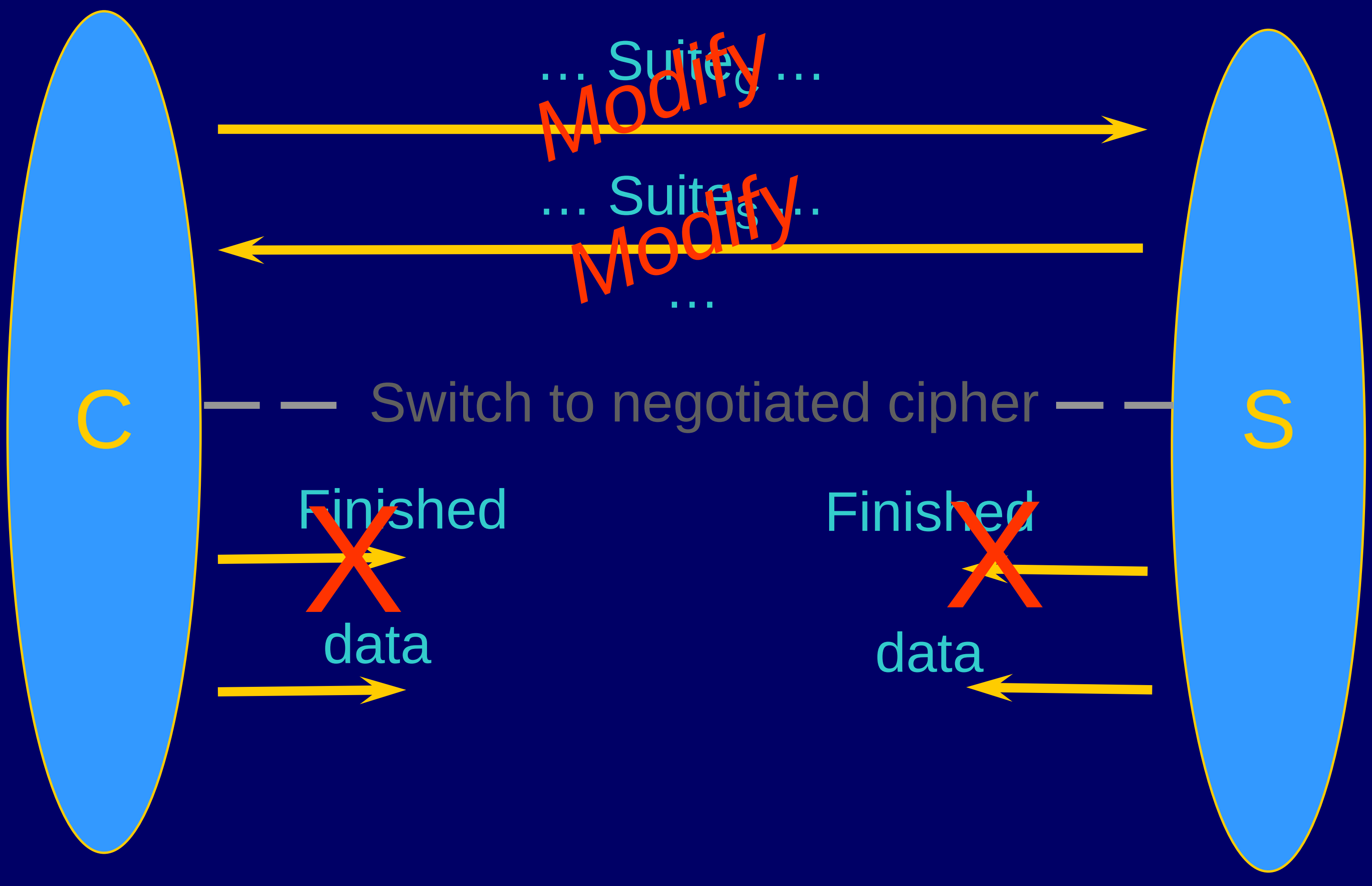
"Core" SSL Handshake with server auth (only)



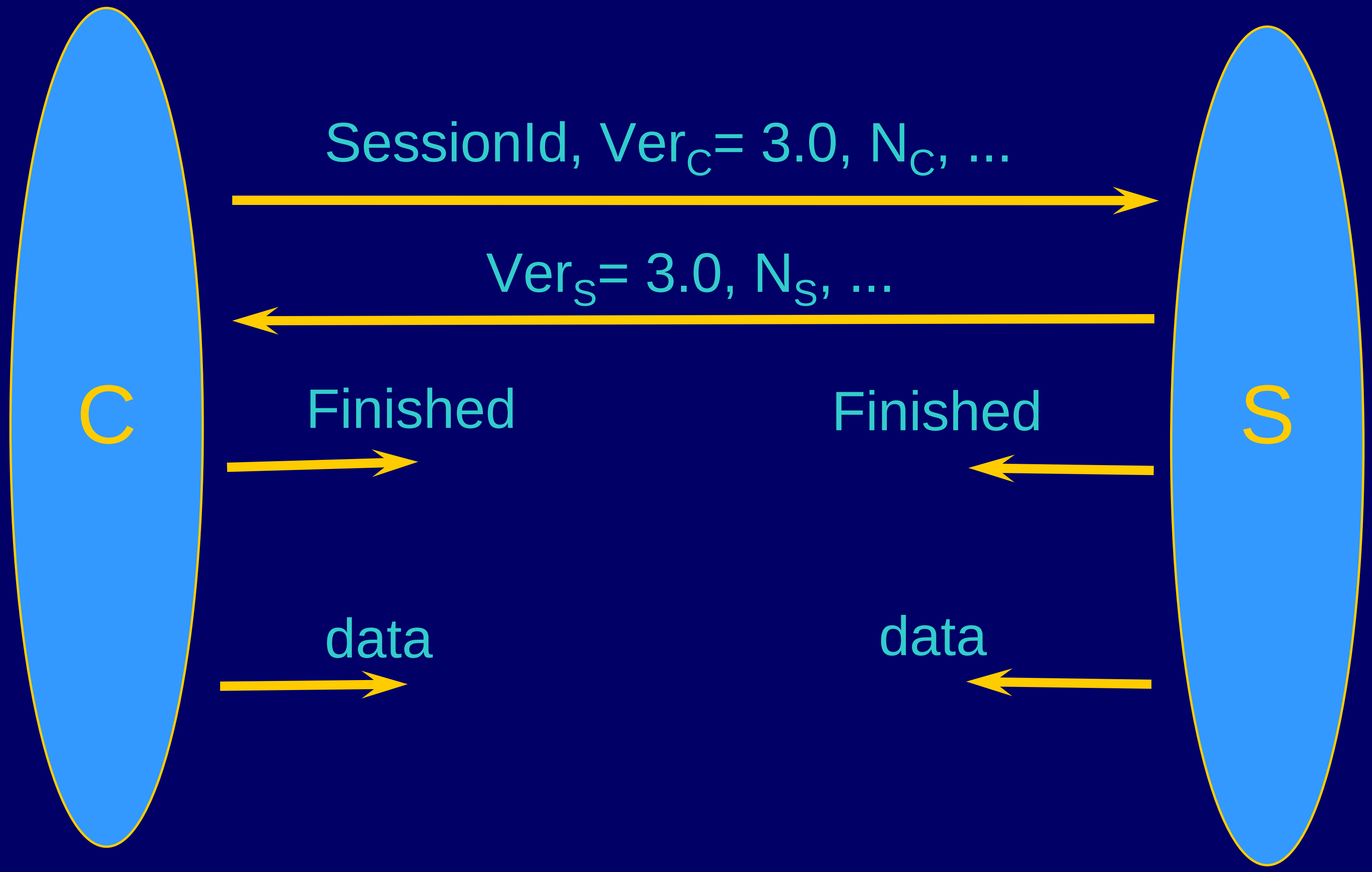
Anomaly (Protocol F)



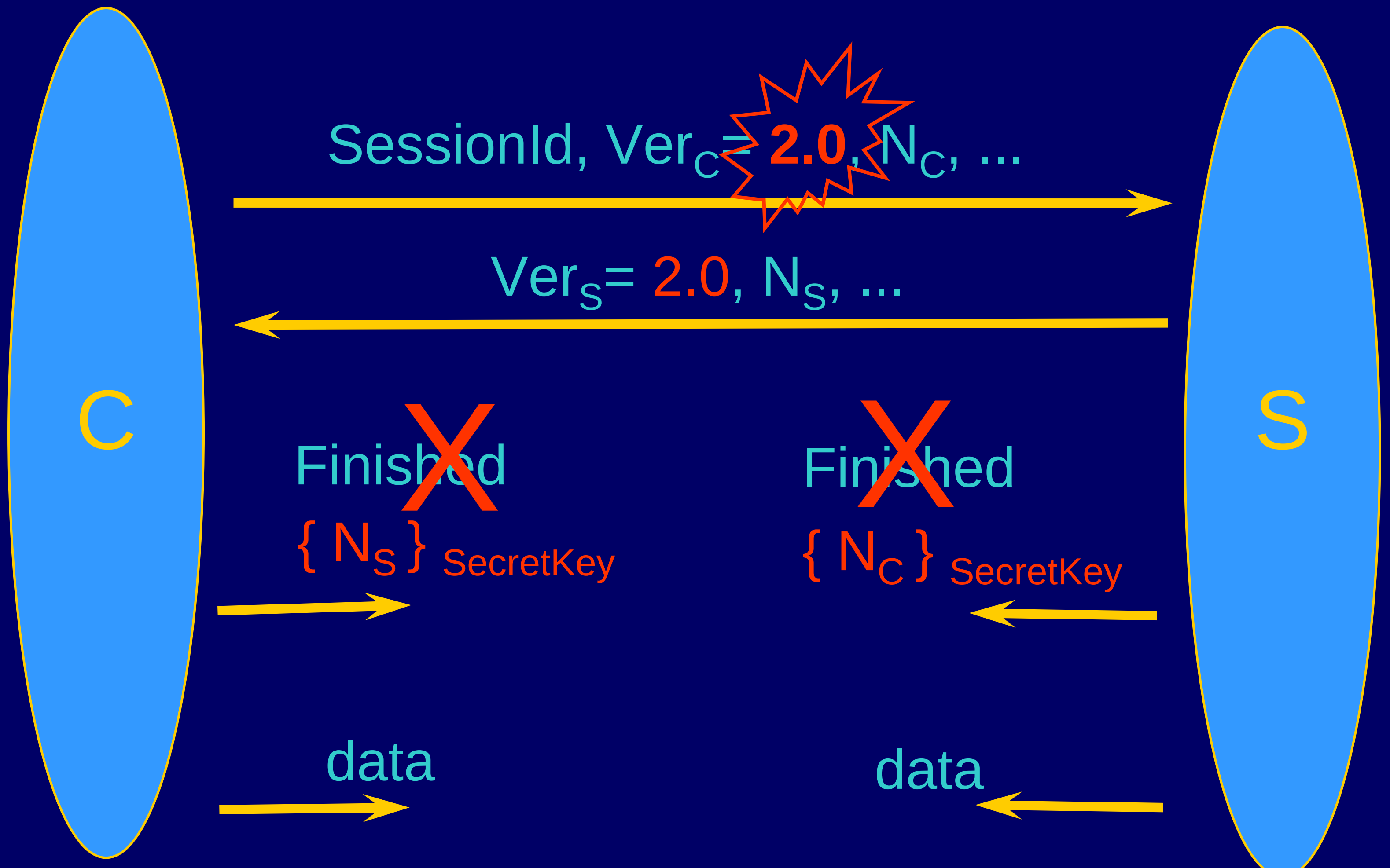
Anomaly (Protocol F)



Protocol Resumption



Version Rollback Attack



SSL 2.0 Finished messages do not include version numbers or cryptosuites

Basic Pattern for Doing Your Project

- Read and understand protocol specification
  - Typically an RFC or a research paper
  - We'll put a few on the website: take a look!
- Choose a tool
  - Murφ by default, but we'll describe many other tools
  - Play with Murφ now to get some experience (installing, running simple models, etc.)
- Start with a simple (possibly flawed) model
  - Rational reconstruction is a good way to go
- Give careful thought to security conditions

Background Reading on SSL 3.0

Optional, for deeper understanding of SSL / TLS

- D. Wagner and B. Schneier. “Analysis of the SSL 3.0 protocol.” USENIX Electronic Commerce ’96.
  - Nice study of an early proposal for SSL 3.0
- J.C. Mitchell, V. Shmatikov, U. Stern. “Finite-State Analysis of SSL 3.0”. USENIX Security ’98.
  - Murφ analysis of SSL 3.0 (similar to this lecture)
  - Actual Murφ model available
- D. Bleichenbacher. “Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1”. CRYPTO ’98.
  - Cryptography is <u>not</u> perfect: this paper breaks SSL 3.0 by directly attacking underlying implementation of RSA