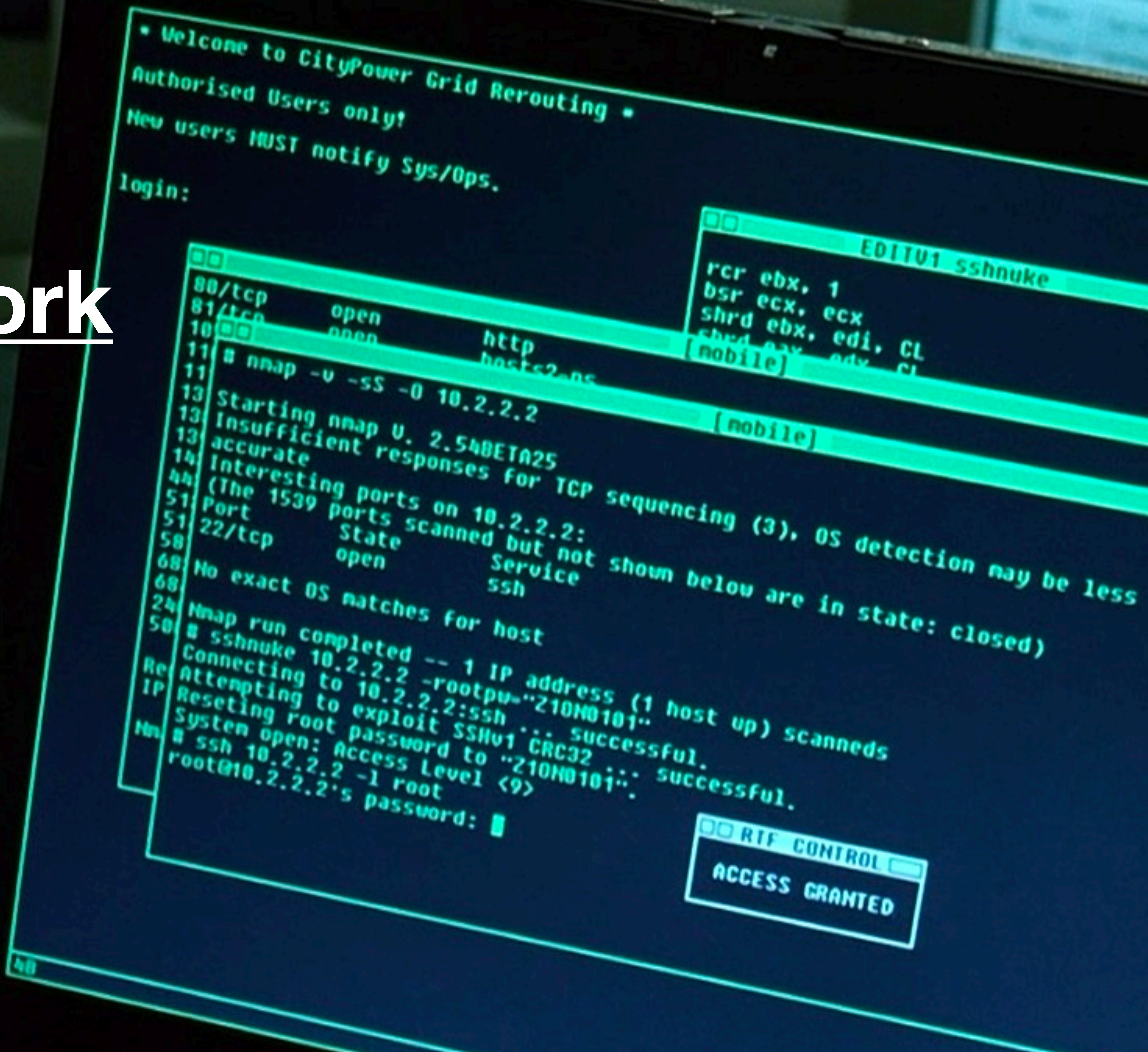
Computer Network Security

ECE 4112/6612

CS 4262/6262

Prof. Frank Li



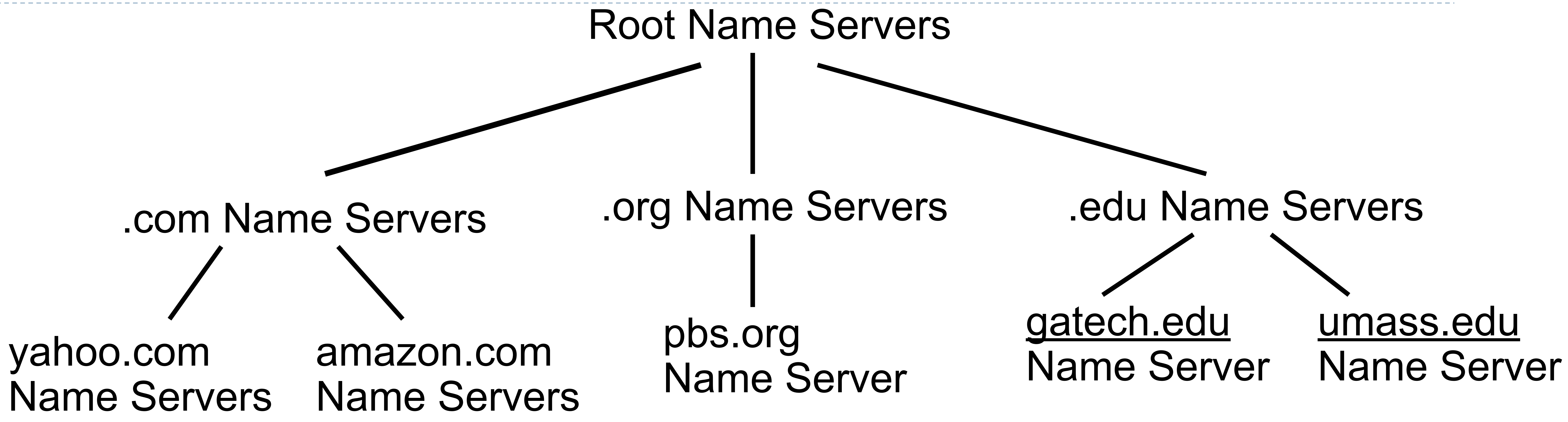
Logistics

Quiz 1 scores released. Regrade requests open until next **Tuesday, Oct 3, midnight.**

Project proposals due this **Friday, midnight**.

Continuing with DNS

Distributed, Hierarchical Database

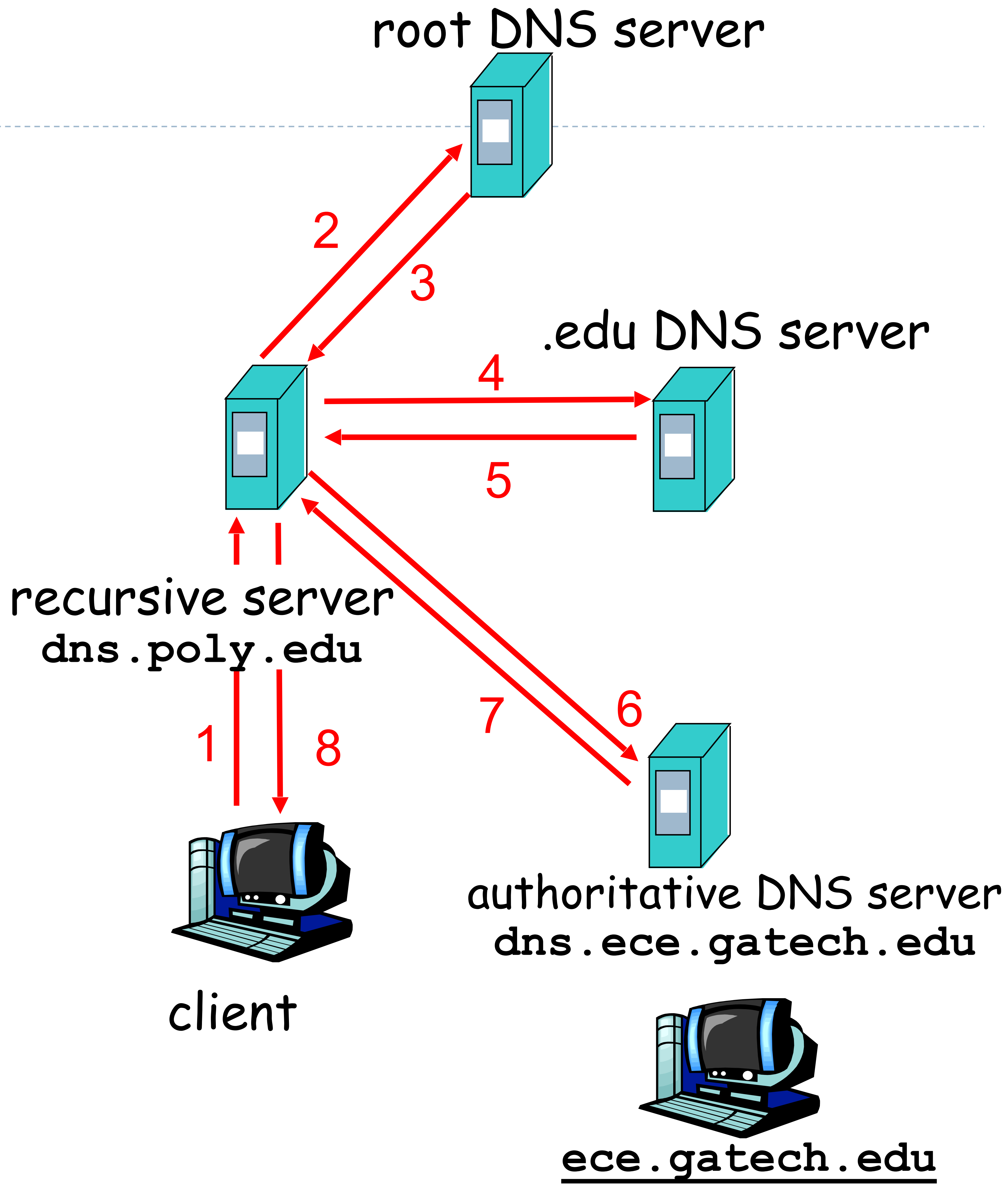


Client wants IP for www.amazon.com; 1st approx:

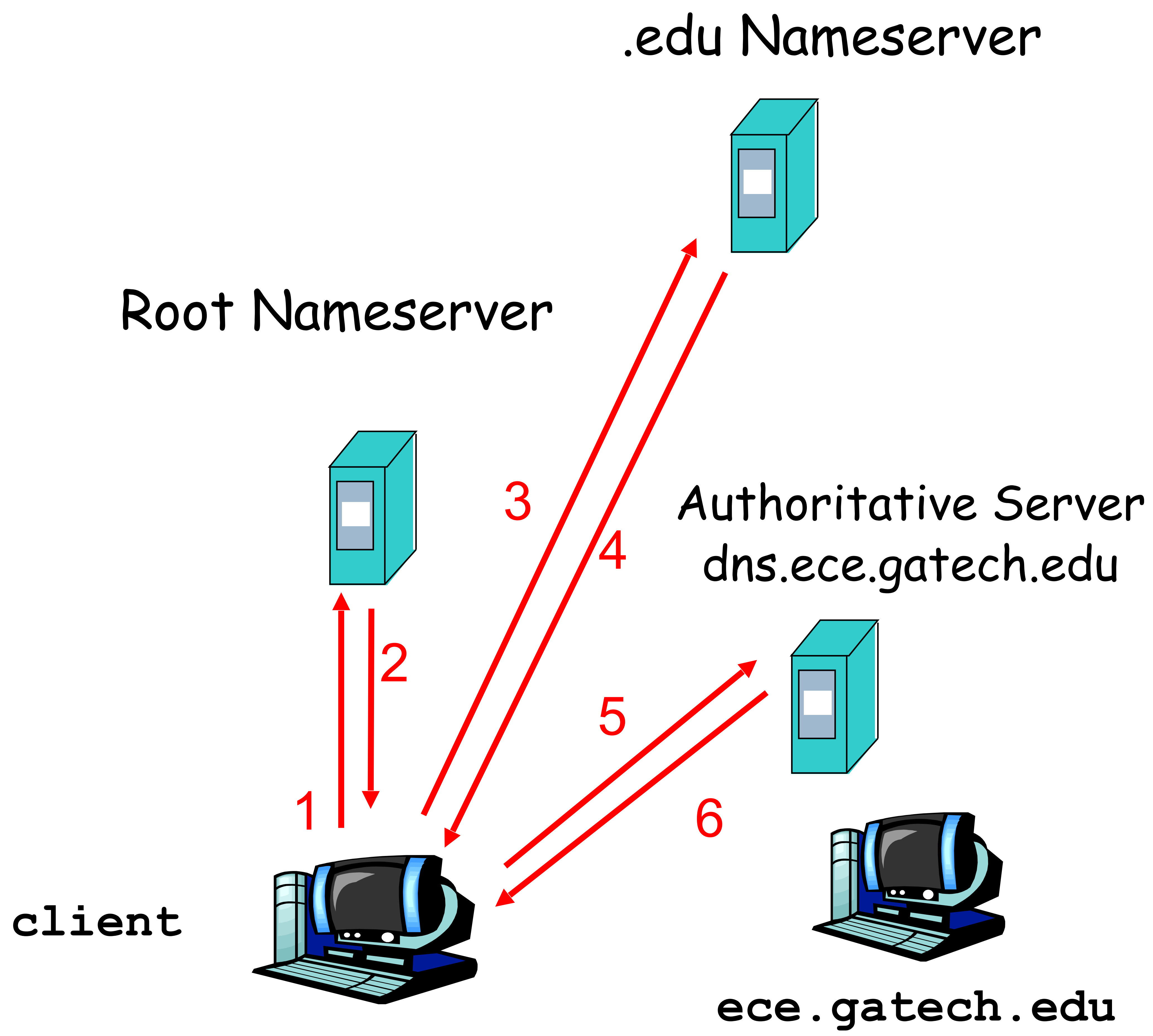
- Client queries a root name server to find "com" name server
- Client queries "com" name server to get amazon.com name server
- Client queries amazon.com name server to get IP address for www.amazon.com

Recursive Query

- Host at ex.poly.edu wants IP address for ece.gatech.edu
- Host sends a "**recursion-requested**" query request to dns.poly.edu.
- Local DNS server does a "**recursive**" search. This requires contacting several other DNS servers before the final answer is given to host.



Iterative Query



DNS: caching and updating records

- once (any) name server learns mapping, it caches mapping
  - cache entries timeout (disappear) after some time. Sometimes based on ttl but *not always*
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited

Inserting records into DNS

- Example: just created startup "Network Utopia"
- Register name networkuptopia.com at a registrar (e.g., Network Solutions)
  - Need to provide registrar with names and IP addresses of your authoritative name server
  - Registrar inserts two RRs into the com TLD server:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```

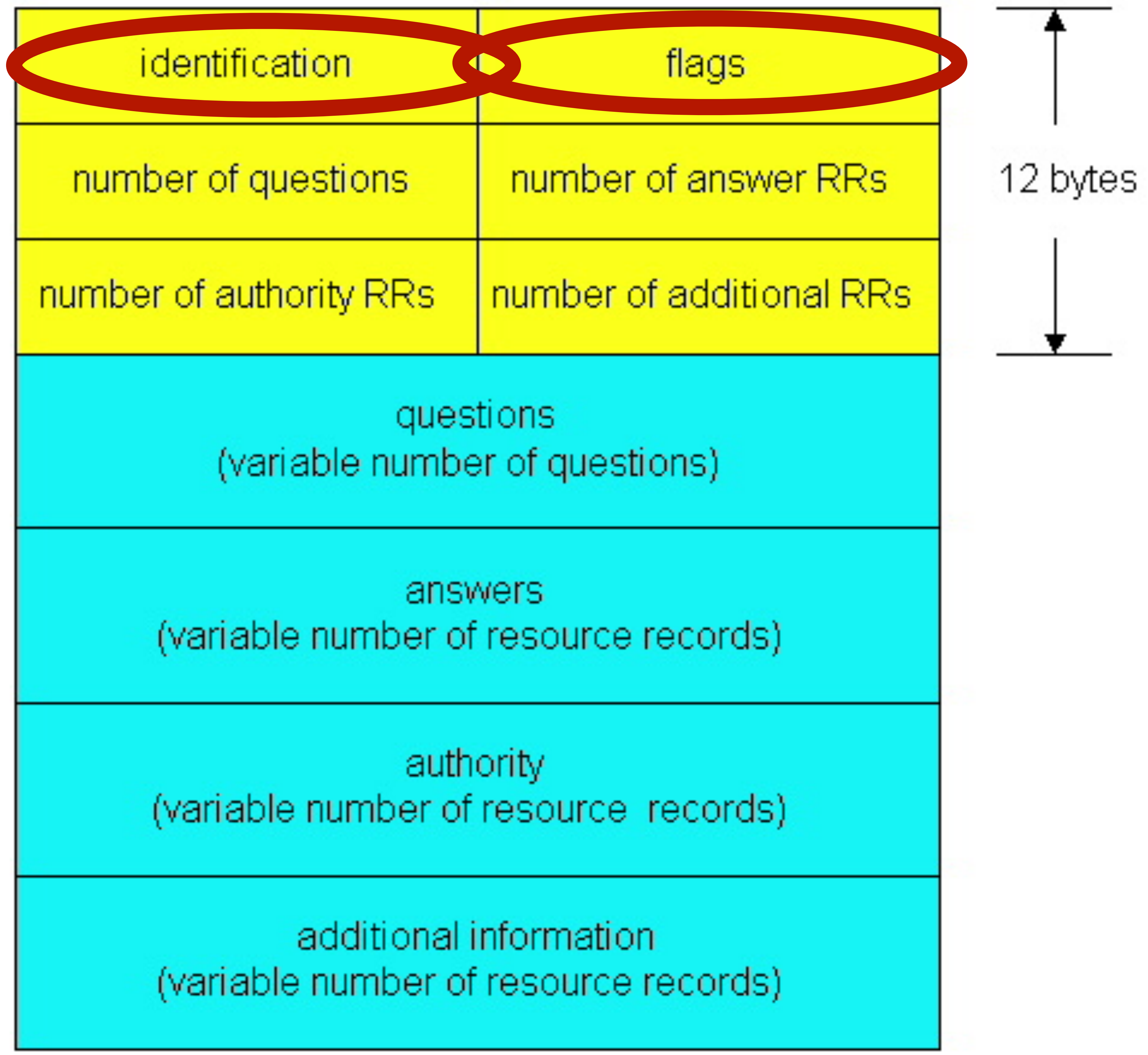
- At authoritative server, can configure Type A record for www.networkuptopia.com and Type MX record for networkutopia.com

DNS protocol, messages

DNS protocol: UDP (port 53) query and reply messages, both with same message format

msg header

- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.
mit.edu.

;; ADDITIONAL SECTION
STRAWB.mit.edu.
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

In general, a single *Resource Record* (RR) like this includes, left-to-right, a DNS name, a *time-to-live*, a family (`IN` for our purposes - ignore), a type (`A` here, which stands for "Address"), and an associated value

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode
;; flags: qr rd ra; QU                                     IONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

"**Answer**" tells us the IP address associated with `eecs.mit.edu` is `18.62.1.6` and we can *cache* the result for 21,600 seconds

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-API
;; global options: +cn
;; Got answer:
;; ->>HEADER<<- opcode
;; flags: qr rd ra; QU                                              3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.          21600    IN       A        18.62.1.6

;; AUTHORITY SECTION:
mit.edu.               11088    IN       NS       BITSY.mit.edu.
mit.edu.               11088    IN       NS       W20NS.mit.edu.
mit.edu.               11088    IN       NS       STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.        126738   IN       A        18.71.0.151
BITSY.mit.edu.         166408   IN       A        18.72.0.3
W20NS.mit.edu.         126738   IN       A        18.70.0.160
```

"**Authority**" tells us the *name servers* responsible for the answer. Each RR gives the *hostname* of a different name server ("`NS`") for names in `mit.edu.` We should cache each record for 11,088 seconds.

If the "**Answer**" had been empty, then the resolver's next step would be to send the original query to one of these name servers.

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION
eecs.mit.edu.

;; AUTHORITY SECT
mit.edu.                11088    IN        NS        BITSY.mit.edu.
mit.edu.                11088    IN        NS        W20NS.mit.edu.
mit.edu.                11088    IN        NS        STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738   IN        A         18.71.0.151
BITSY.mit.edu.          166408   IN        A         18.72.0.3
W20NS.mit.edu.          126738   IN        A         18.70.0.160
```

"**Additional**" provides extra information to save us from making separate lookups and helps with bootstrapping (**glue records**).

Here, it tells us the IP addresses for the hostnames of the name servers. We add these to our cache.

DNS Security Threats

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.           21

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

What if the `mit.edu` name server is untrustworthy? Could its operator steal, say, all of our web surfing to Facebook?

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.           21                              .1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

Let’s look at a flaw in the original DNS design (since fixed)

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.            21600   IN       A        18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                 11088   IN       NS       BITSY.mit.edu.
mit.edu.                 11088   IN       NS       W20NS.mit.edu.
mit.edu.                 11088   IN       NS       www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com         11088   IN       A        18.6.6.6
BITSY.mit.edu.           166408  IN       A        18.72.0.3
W20NS.mit.edu.           126738  IN       A        18.70.0.160
```

What could happen if the mit.edu name server returns the following to us instead?

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                    IN      A

;; ANSWER SECTION:
eecs.mit.edu.

;; AUTHORITY SECTION:
mit.edu.                 11088    IN      NS      BITSY.mit.edu.
mit.edu.                 11088    IN      NS      W20NS.mit.edu.
mit.edu.                 11088    IN      NS      www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com         11088    IN      A       18.6.6.6
BITSY.mit.edu.           166408   IN      A       18.72.0.3
W20NS.mit.edu.           126738   IN      A       18.70.0.160
```

We'd dutifully store in our cache a mapping of `www.facebook.com` to an IP address under MIT's control. (It could have been any IP address they wanted, not just one of theirs.)

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.

;; AUTHORITY SECTION:
mit.edu.                                                        edu.
mit.edu.                 11088    IN        NS        W20NS.mit.edu.
mit.edu.                 11088    IN        NS        www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com         11088    IN        A         18.6.6.6
BITSY.mit.edu.           166408   IN        A         18.72.0.3
W20NS.mit.edu.           126738   IN        A         18.70.0.160
```

Next time one of our clients starts to connect to `www.facebook.com`, it will ask our resolver for the corresponding IP address. The resolver will find the answer in its cache and return **18.6.6.6** 😟

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com        11088   IN      A       18.6.6.6
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

How do we fix such ***cache poisoning***?

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-AP
;; global options: +c
;; Got answer:
;; ->>HEADER<<- opcod
;; flags: qr rd ra; Q

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.

;; AUTHORITY SECTION:
mit.edu.                 11088   IN     NS     BITSY.mit.edu.
mit.edu.                 11088   IN     NS     W20NS.mit.edu.
mit.edu.                 11088   IN     NS     www.facebook.com.

;; ADDITIONAL SECTION:
www.facebook.com         11088   IN     A      18.6.6.6
BITSY.mit.edu.           166408  IN     A      18.72.0.3
W20NS.mit.edu.           126738  IN     A      18.70.0.160
```

Don't accept **Additional** records unless they're for the domain of the name server we queried

E.g., contacting a name server for `mit.edu` ⇒ only accept additional records from `*.mit.edu`

No extra risk in accepting these since server could return them to us directly in an **Answer** anyway.

dig eecs.mit.edu A

```
; ; <<>> DiG 9.6.0-AP
;; global options: +c
;; Got answer:
;; ->>HEADER<<- opcod
;; flags: qr rd ra; Q

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu.

;; AUTHORITY SECTION:
mit.edu.                11088   IN   NS   BITSY.mit.edu
mit.edu.                11
mit.edu.                11

;; ADDITIONAL SECTION:
www.facebook.com        11
BITSY.mit.edu.          16
W20NS.mit.edu.          12
```

Don’t accept **Additional** records unless they’re for the domain of the name server we queried

E.g., contacting a name server for `mit.edu` ⇒ only accept additional records from `*.mit.edu`

No extra risk in accepting these since server could return them to us directly in an **Answer** anyway.

This is called “bailiwick checking”.

bailiwick | ˈbāləˌwik |

noun

1 (**one's bailiwick**) one's sphere of operations or particular area of interest: *you never give the presentations—that's my bailiwick.*

Bailiwick Checking

Bailiwick rules (for querying for www.google.com):

1. Root servers can return any record (bailiwick is everything)
   - Ex: k.root-server.net is a root server. It responds with a.gtld-servers.net as one of the .com name servers (in the authority + additional/glue section).
2. .com NS can return any record within .com (bailiwick is .com)
   - Ex: a.gtld-server.net returns ns1.google.com as a google.com name server (in the authority and additional)
3. google.com NS can return any record within google.com (bailiwick is google.com)
   - Ex: Returns answer for www.google.com

Modern DNS software enforces bailiwick checking (not in original DNS design though)

DNS Threats: Spoofing

- If an attacker observes the DNS identification number, they can spoof a DNS response to a victim, giving them false DNS answers (e.g., respond to a DNS query for `mail.google.com` with an A record pointing to the attacker server)

- Has been used in real attacks, including censorship.

16 bits	*16 bits*
SRC=53	DST=53
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

DNS Threats: Spoofing

What about *blind spoofing*?

- Say we look up `mail.google.com`; how can an **off-path** attacker feed us a bogus `A` answer before the legitimate server replies?
- How can such a **remote** attacker even know we are looking up `mail.google.com`?

Suppose, e.g., we visit a web page under their control:

`...<img src="http://mail.google.com" …> ...`

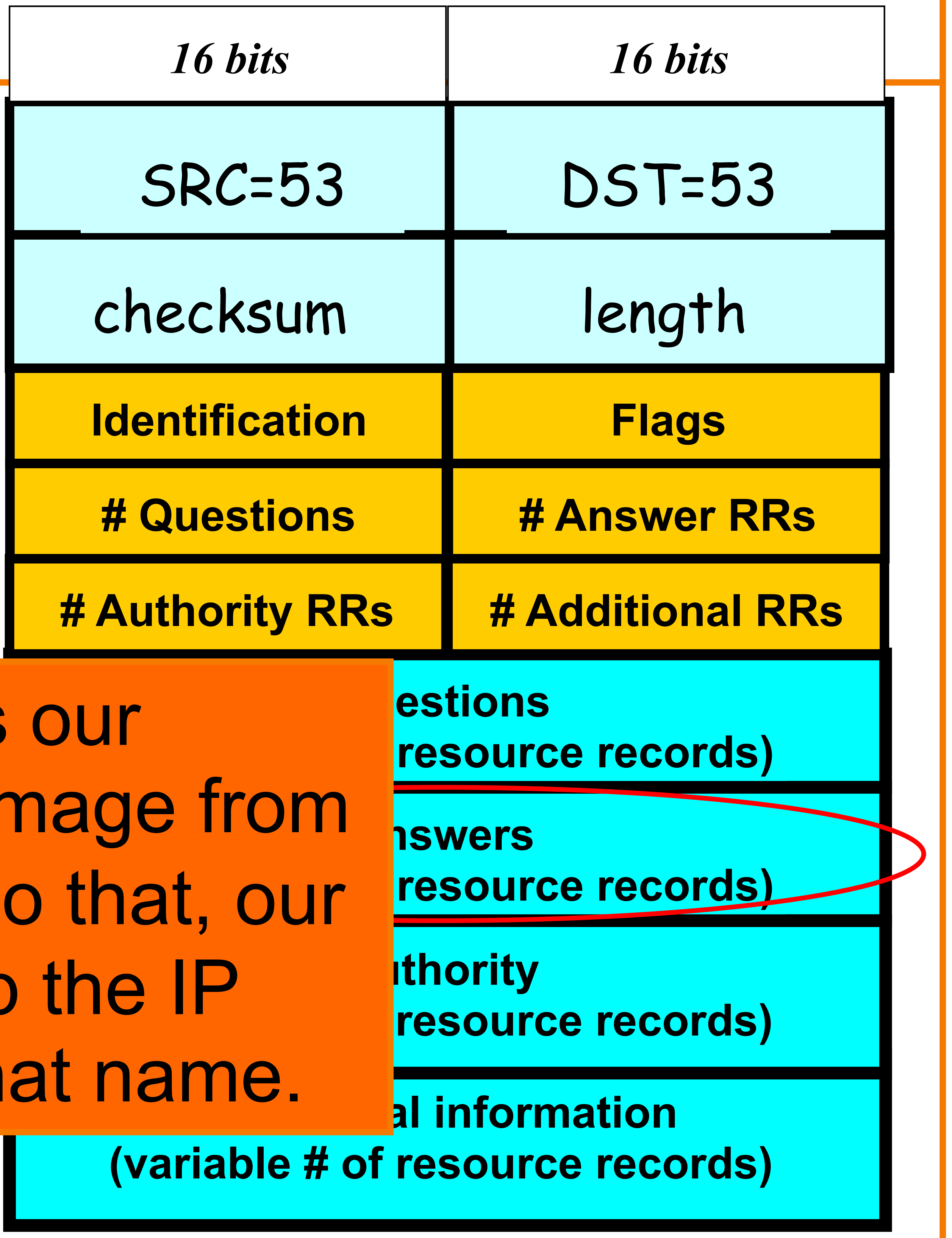
16 bits	*16 bits*
SRC=53	DST=53
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

DNS Threats: Spoofing

What about *blind spoofing*?

- Say we look up `mail.google.com`; how can an **off-path** attacker feed us a bogus `A` answer before the legitim
- How ca even k `mail.`

This HTML snippet causes our browser to try to fetch an image from `mail.google.com`. To do that, our browser first has to look up the IP address associated with that name.



Suppose, e.g., we visit a web page under their control:

`...<img src="http://mail.google.com" …> ...`

DNS Blind Spoofing, con't

Fix?

Once they know we're looking it up, they just have to guess the Identification field, and reply before legit server.

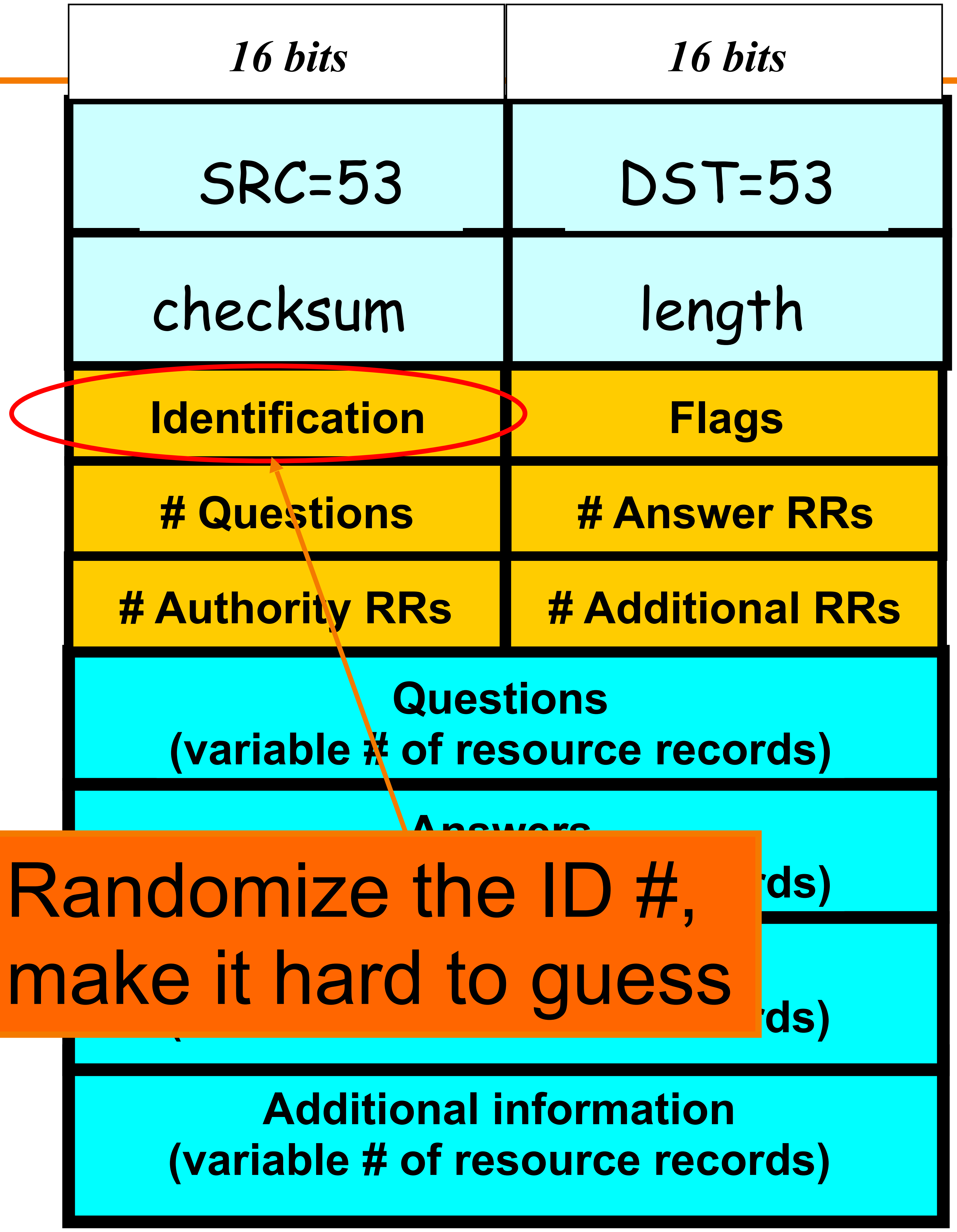
How hard is that?

Originally, identification field incremented by 1 for each request. How does attacker guess it?

(Assuming attacker controls their domain's name server)

<img src="http://attacker.com" …> They observe ID k here

<img src="http://mail.google.com" …> So this will be k+1



Randomize the ID #, make it hard to guess

DNS Blind Spoofing, con't

Once we randomize the Identification, attacker has a 1/65536 chance of guessing it correctly.
Are we pretty much safe?

Attacker can send *lots* of replies, not just one …

However: once a reply from legit server arrives (with correct Identification), it's **cached** and no more opportunity to poison it. Victim is inoculated!

16 bits	16 bits
SRC=53	DST=53
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

Unless attacker can send 1000s of replies before legit arrives, we're likely safe - phew! **?**

DNS Blind Spoofing (Kaminsky 2008)

- Two key ideas:
  - Spoof uses **Additional** field (rather than **Answer**)
  - Attacker can get around caching of legit replies by generating a series of *different* name lookups:

```
<img src="http://random1.google.com" …>
<img src="http://random2.google.com" …>
<img src="http://random3.google.com" …>
                 . . .
<img src="http://randomN.google.com" …>
```

Kaminsky Blind Spoofing, con't

For each lookup of *randomk*.google.com, attacker **spoofs** a bunch of records like this, each with a different Identifier

```
;; QUESTION SECTION:
;randomk.google.com.                    IN      A

;; ANSWER SECTION:
randomk.google.com        21600         IN      A        doesn't matter

;; AUTHORITY SECTION:
google.com.               11088         IN      NS       mail.google.com

;; ADDITIONAL SECTION:
mail.google.com           126738        IN      A        6.6.6.6
```

Once they win the race, not only have they poisoned `mail.google.com` …

Kaminsky Blind Spoofing, con't

For each lookup of *randomk*.google.com, attacker **spoofs** a bunch of records like this, each with a different Identifier

```
;; QUESTION SECTION:
;randomk.google.com.                IN      A

;; ANSWER SECTION:
randomk.google.com        21600     IN      A       doesn't matter

;; AUTHORITY SECTION:
google.com.               11088     IN      NS      mail.google.com

;; ADDITIONAL SECTION:
mail.google.com           126738    IN      A       6.6.6.6
```

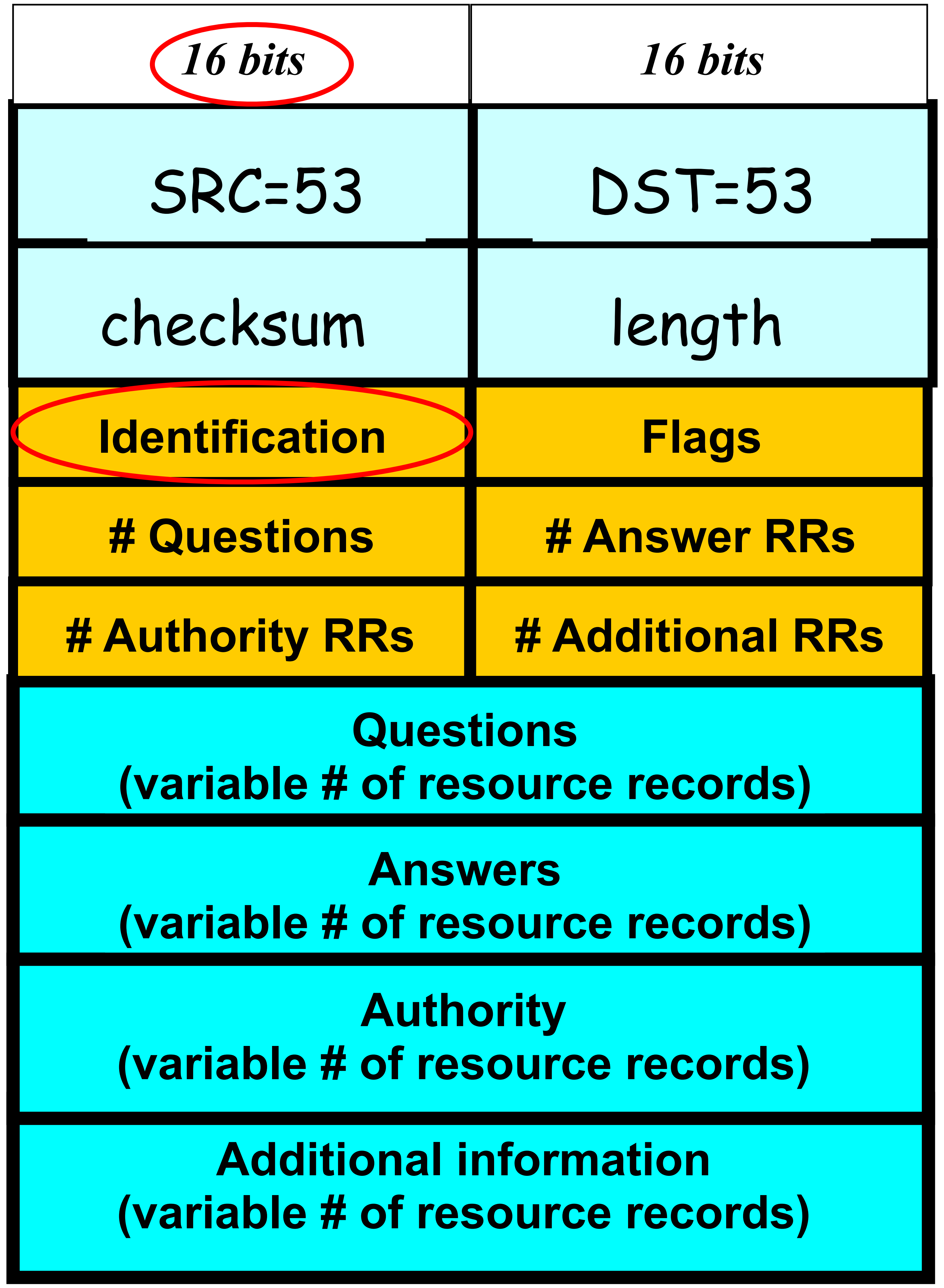
Once they win the race, not only have they poisoned mail.google.com … but also the cached NS record for google.com's name server - so any **future** *X*.google.com lookups *go through the attacker's machine*

Defending Against Blind Spoofing

Central problem: all that tells a client they should accept a response is that it matches the Identification field.

With only 16 bits, it lacks sufficient entropy: even if truly random, the *search space* an attacker must *brute force* is too small.

Where can we get more entropy?



Defending Against Blind Spoofing

Central problem: all that tells a client they should accept a response is that it matches the Identification field.

With only 16 bits, it lacks sufficient entropy: even if truly random, the *search space* an attacker must *brute force* is too small.

Where can we get more entropy? (*Without* requiring a protocol change.)

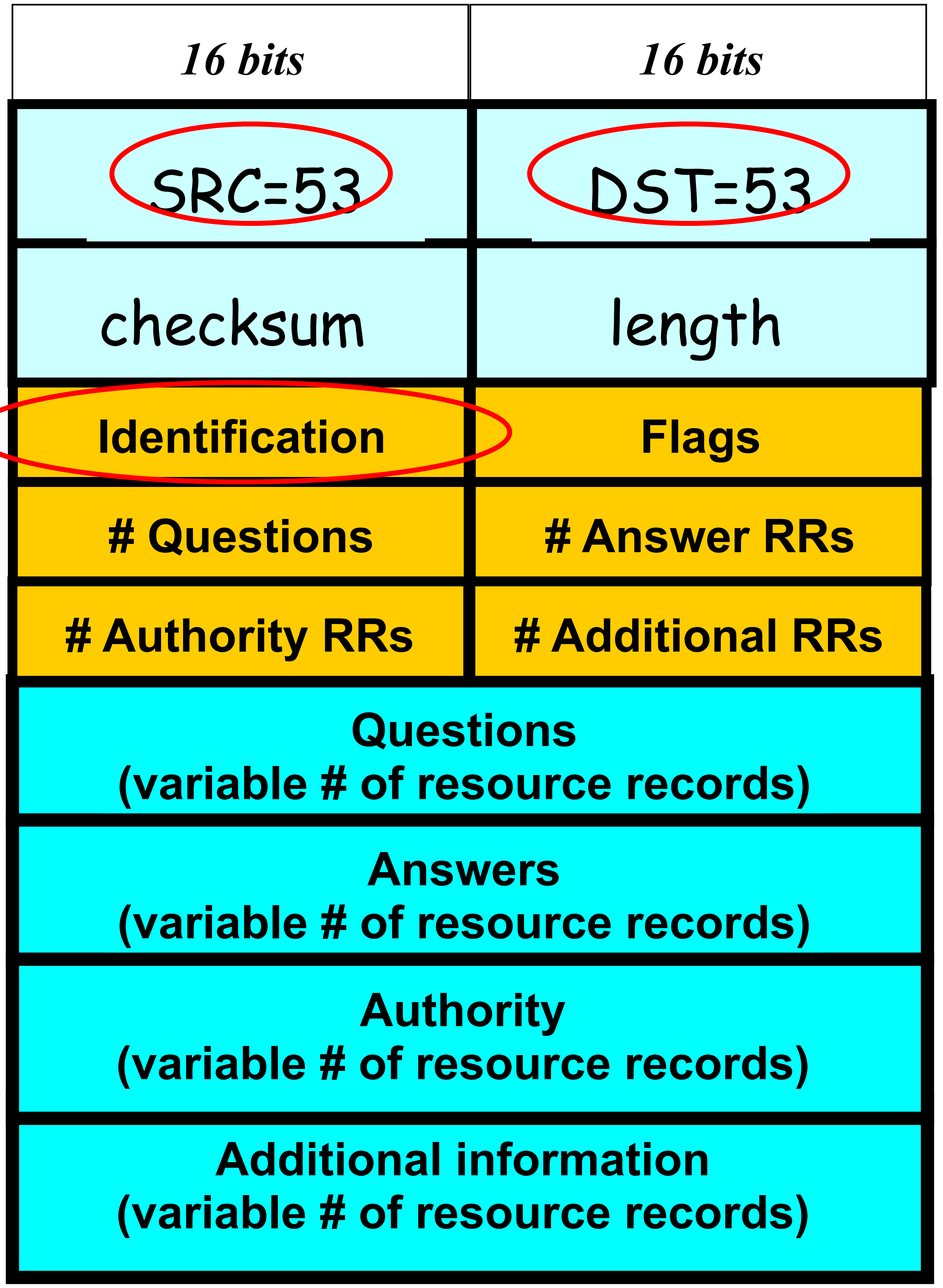
16 bits	*16 bits*
SRC=53	DST=53
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

Defending Against Blind Spoofing

For requestor to receive DNS reply, needs both correct Identification and correct ports.

On a request, DST port = 53. SRC port usually also 53 - but not fundamental, just convenient.

Total *entropy*: 16 bits



Defending Against Blind Spoofing

Total *entropy*: ? bits

"Fix": client uses random source port ⇒ attacker doesn't know correct dest. port to use in reply

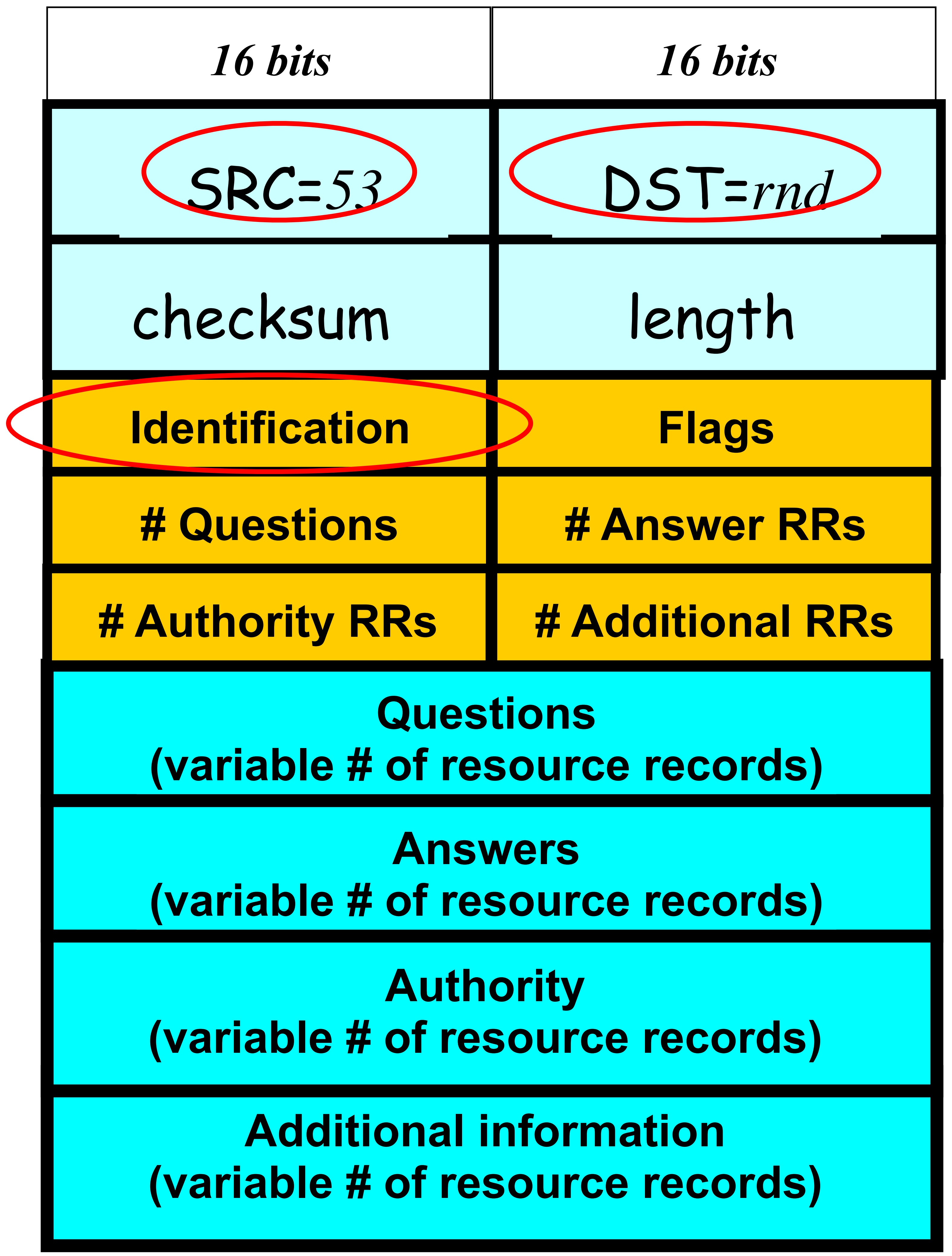
16 bits	*16 bits*
SRC=*53*	DST=*rnd*
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

Defending Against Blind Spoofing

"Fix": client uses random source port ⇒ attacker doesn't know correct dest. port to use in reply

32 bits of entropy makes it orders of magnitude harder for attacker to guess all the necessary fields and dupe victim into accepting spoof response.

Total *entropy*: 32 bits



Defending Against Blind Spoofing

"Fix": client uses random source port ⇒ attacker doesn't know correct dest. port to use in reply

32 bits of entropy makes it orders of magnitude harder for attacker to guess all the necessary fields and dupe victim into accepting spoof response.

This is what primarily "secures" DNS against blind spoofing today. (Note: not all resolvers have implemented random source ports!)

Total *entropy*: 32 bits

16 bits	*16 bits*
SRC=*53*	DST=*rnd*
checksum	length
Identification	**Flags**
# Questions	**# Answer RRs**
# Authority RRs	**# Additional RRs**
Questions (variable # of resource records)	
Answers (variable # of resource records)	
Authority (variable # of resource records)	
Additional information (variable # of resource records)	

DNS Poisoning

Malicious DNS name server could provide invalid authority/additional section for domains outside of its zone, poisoning the DNS cache.

- Prevent this with DNS bailiwick checking (only accept DNS records from a name server that are within its zone)

DNS spoofing requires the attacker to match the DNS ID #.

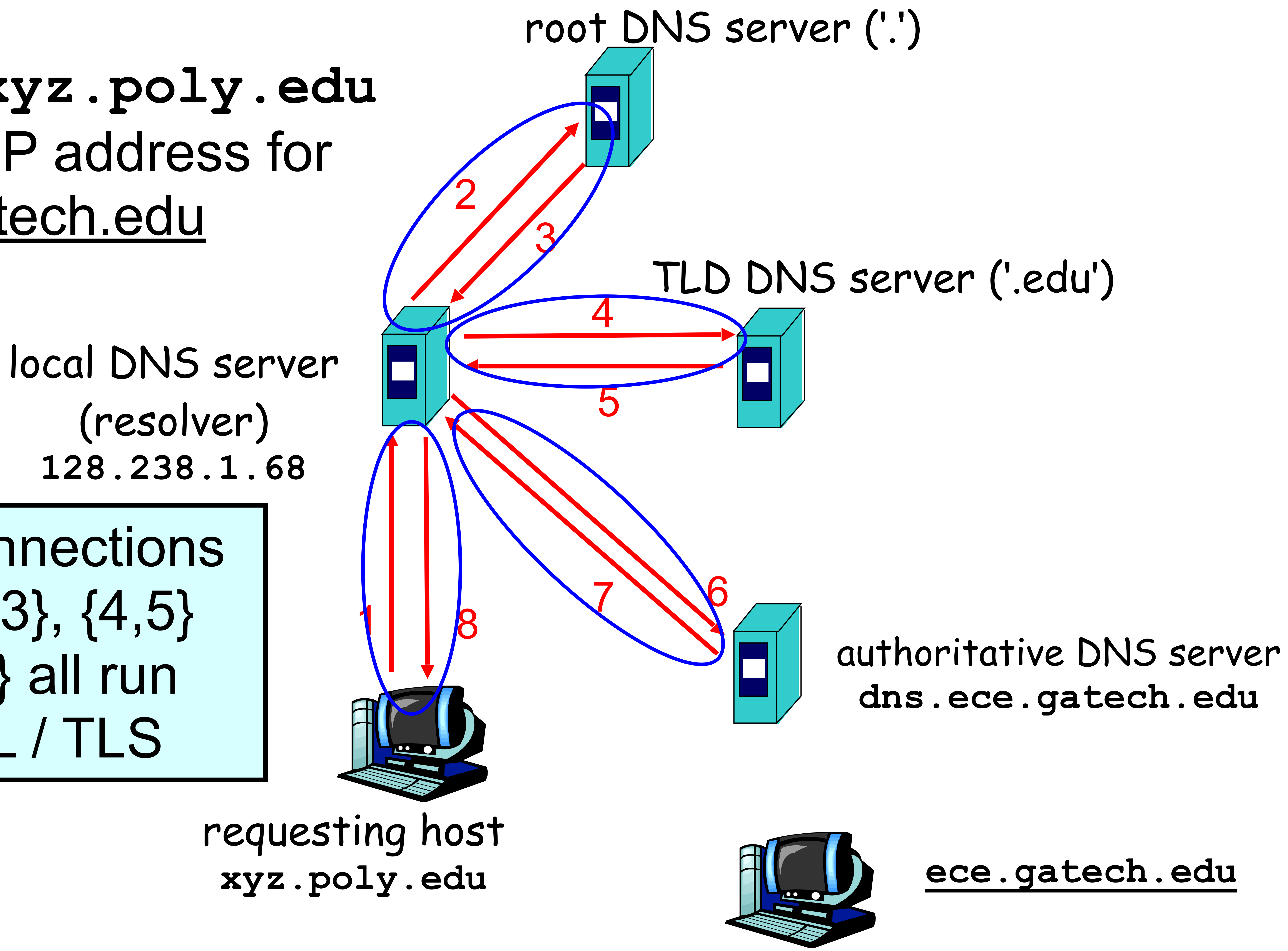
- Need to make ID # random to make this unpredictable.
- Attacker can guess ID # blindly, but must race against real DNS response. Kaminsky attack makes attack effective (try random subdomains, only need to win race once).
    - Made hard by randomizing the request source port also, adding more bits of entropy that need to be guessed correctly.
- Still easy if attacker can observe the DNS request.

Securing DNS Lookups

- How can we ensure when clients look up names with DNS, they can trust answers they receive?
- Idea #1: do DNS lookups over TLS
  - (assuming either we run DNS over TCP, or we use “Datagram TLS”)

Securing DNS using SSL / TLS?

Host at **`xyz.poly.edu`** wants IP address for <u>ece.gatech.edu</u>



Idea: connections {1,8}, {2,3}, {4,5} and {6,7} all run over SSL / TLS

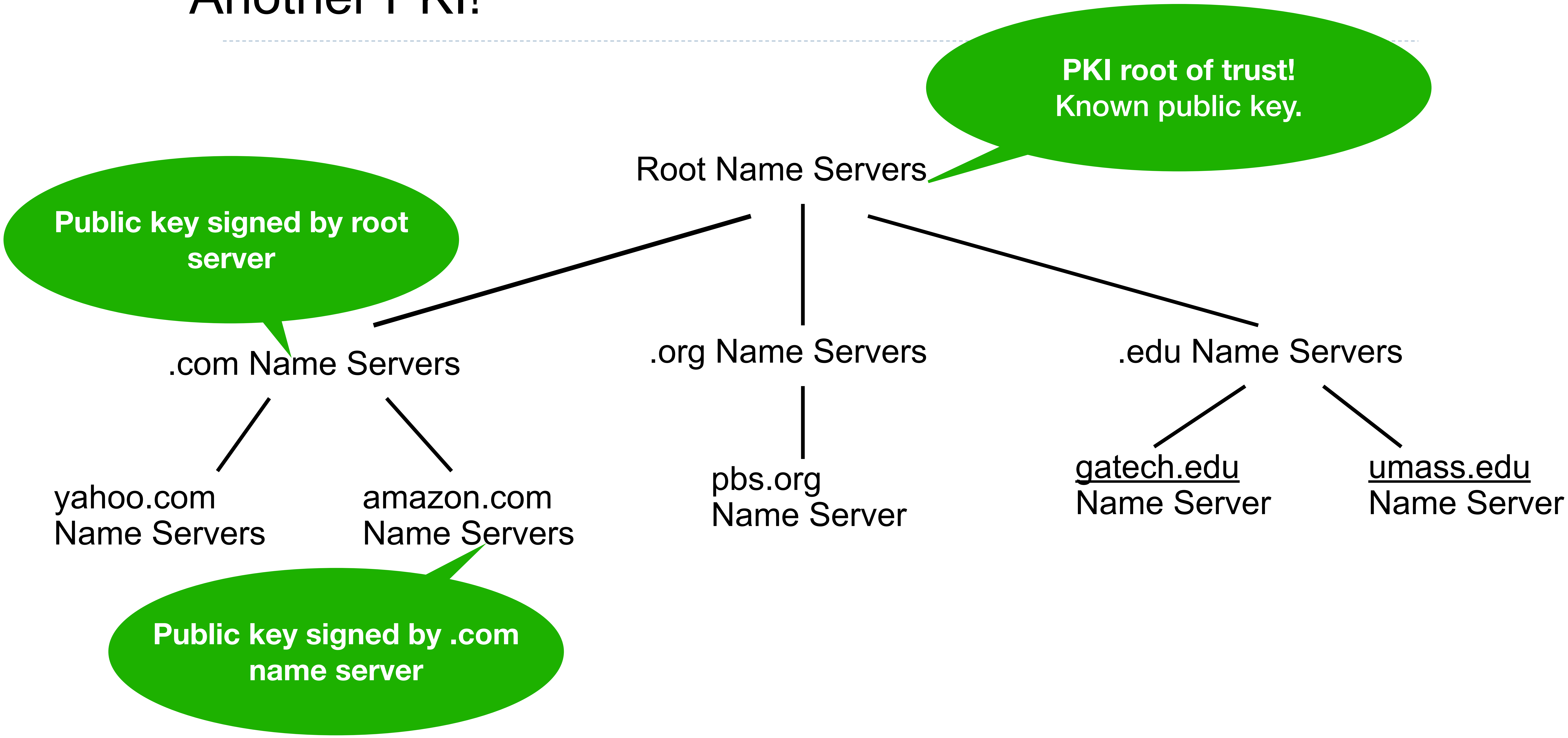
Securing DNS Lookups

- How can we ensure when clients look up names with DNS, they can trust answers they receive?
- Idea #1: do DNS lookups over TLS
  - (assuming either we run DNS over TCP, or we use "Datagram TLS")
  - Issues?
    - Performance: DNS is very lightweight. TLS is not.
    - Caching: crucial for DNS scaling. But then how do we keep authentication assurances?
      - ***Object*** *security vs.* ***Channel*** *security*

Securing DNS Lookups

- How can we ensure when clients look up names with DNS, they can trust answers they receive?
- Idea #1: do DNS lookups over TLS
  - (assuming either we run DNS over TCP, or we use "Datagram TLS")
  - Issues?
    - Performance: DNS is very lightweight. TLS is not.
    - Caching: crucial for DNS scaling. But then how do we keep authentication assurances?
      - ***Object*** *security vs.* ***Channel*** *security*
- Idea #2: make DNS results like *certs*
  - I.e., a verifiable signature that guarantees who generated a piece of data; signing happens **off-line**

Another PKI!



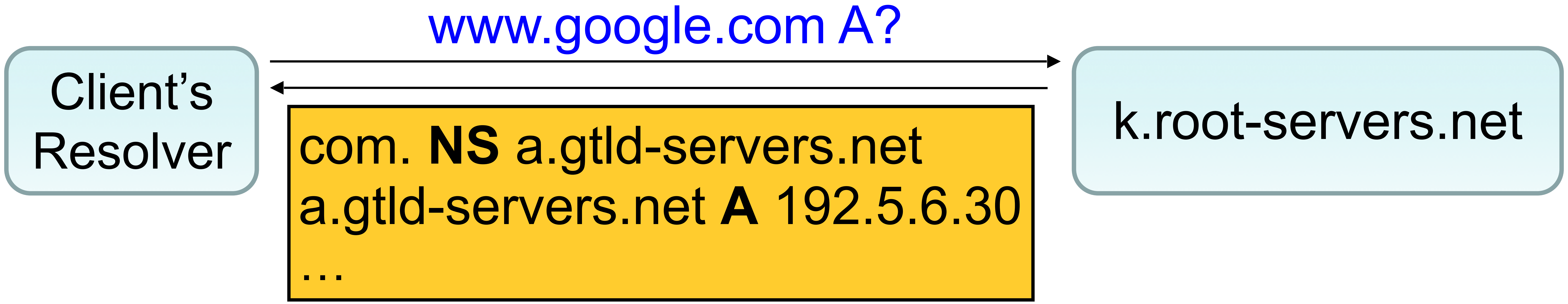
PKI root of trust!
Known public key.
Root Name Servers
Public key signed by root server
.com Name Servers
.org Name Servers
.edu Name Servers
yahoo.com Name Servers
amazon.com Name Servers
pbs.org Name Server
gatech.edu Name Server
umass.edu Name Server
Public key signed by .com name server

Operation of DNSSEC

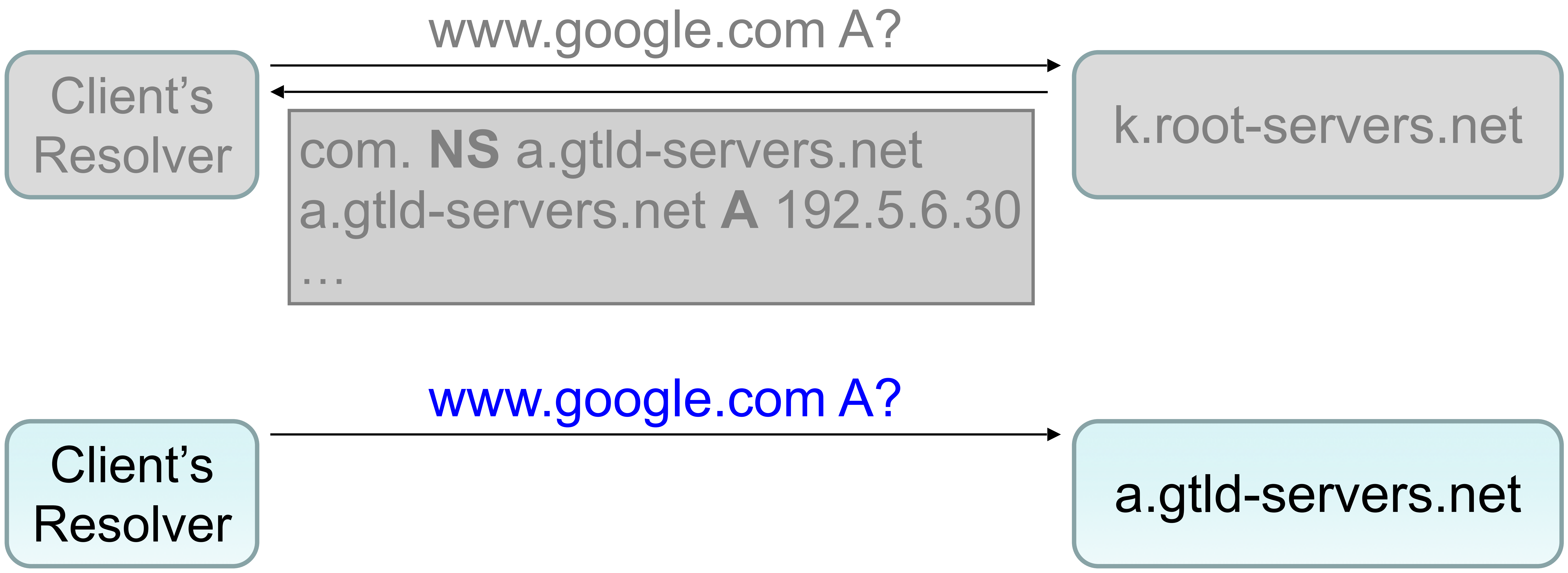
- DNSSEC = standardized DNS security extensions currently being deployed
- As a resolver works its way from DNS root down to final name server for a name, at each level it gets a signed statement regarding the key(s) used by the next level
  - This builds up a chain of trusted keys
  - Resolver has root's key wired into it
- The final answer that the resolver receives is signed by that level's key
  - Resolver can trust it's the right key because of chain of support from higher levels
- All keys *as well as signed results* are *cacheable*

Ordinary DNS:

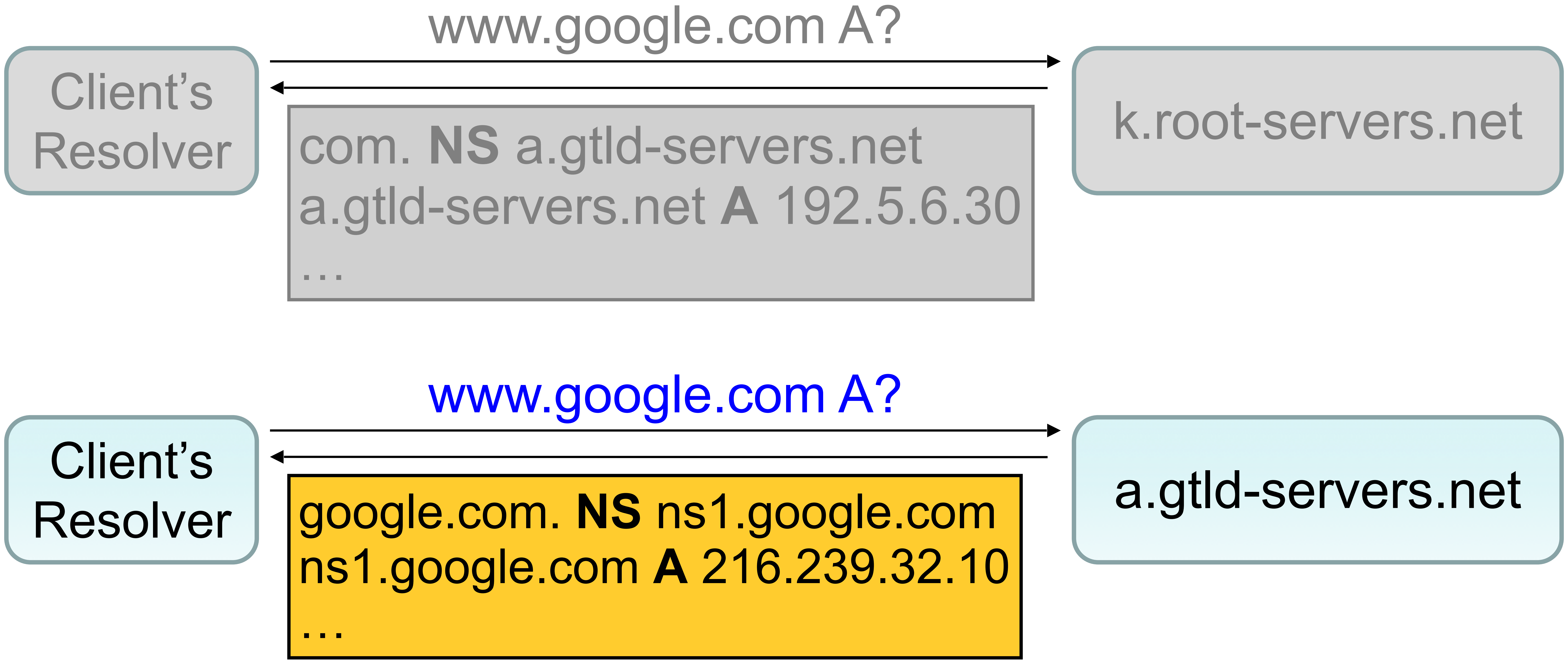
Ordinary DNS:



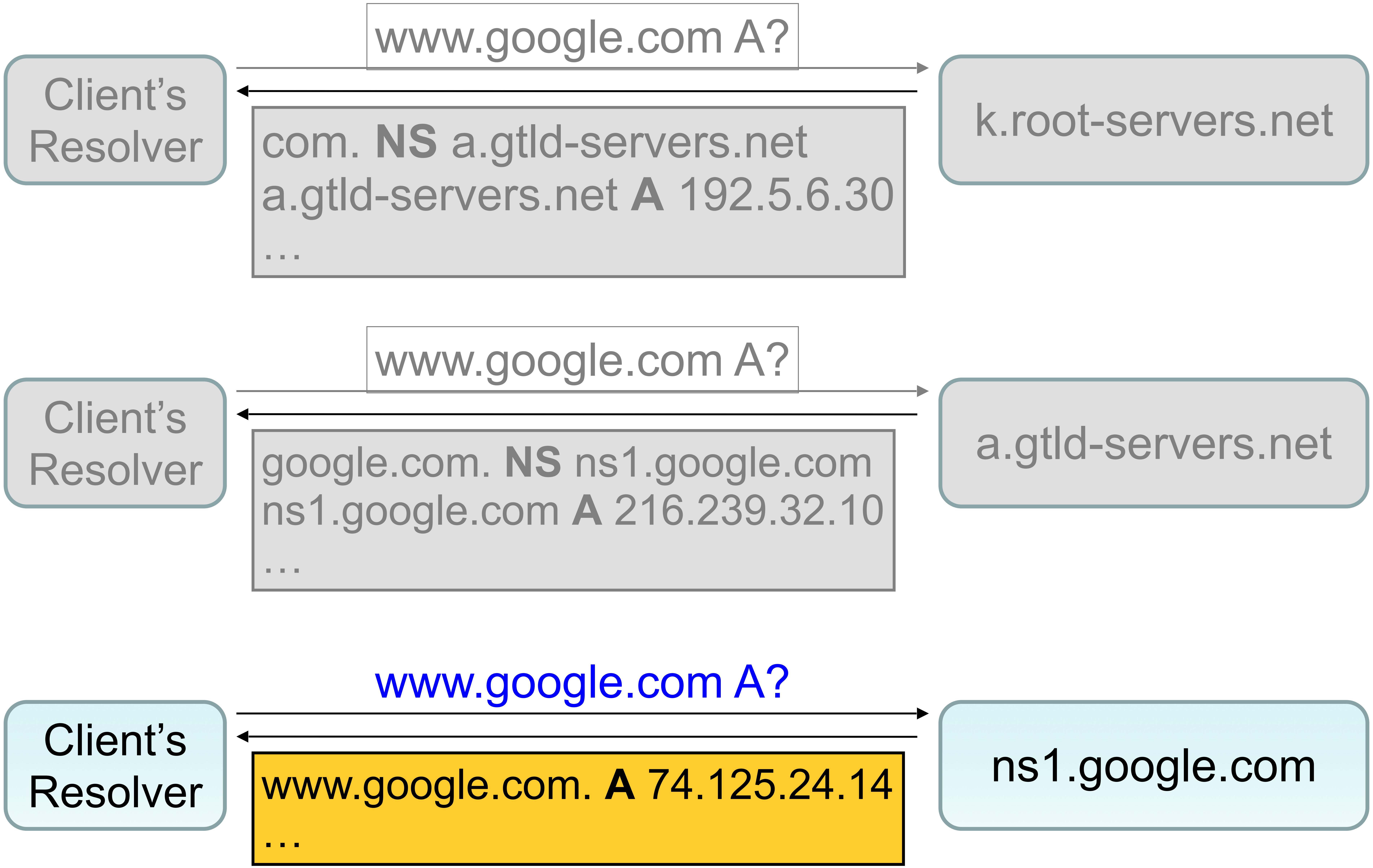
Ordinary DNS:



Ordinary DNS:

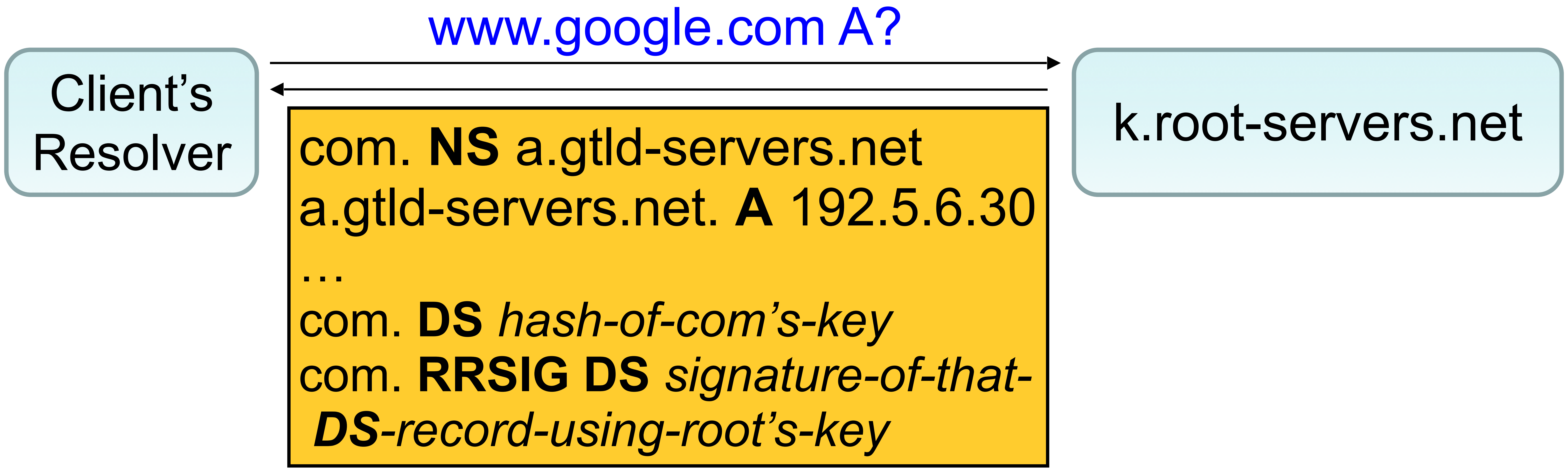


Ordinary DNS:



Ordinary DNS:

DNSSEC (with simplifications):



DNSSEC (with simplifications):

www.google.com A?

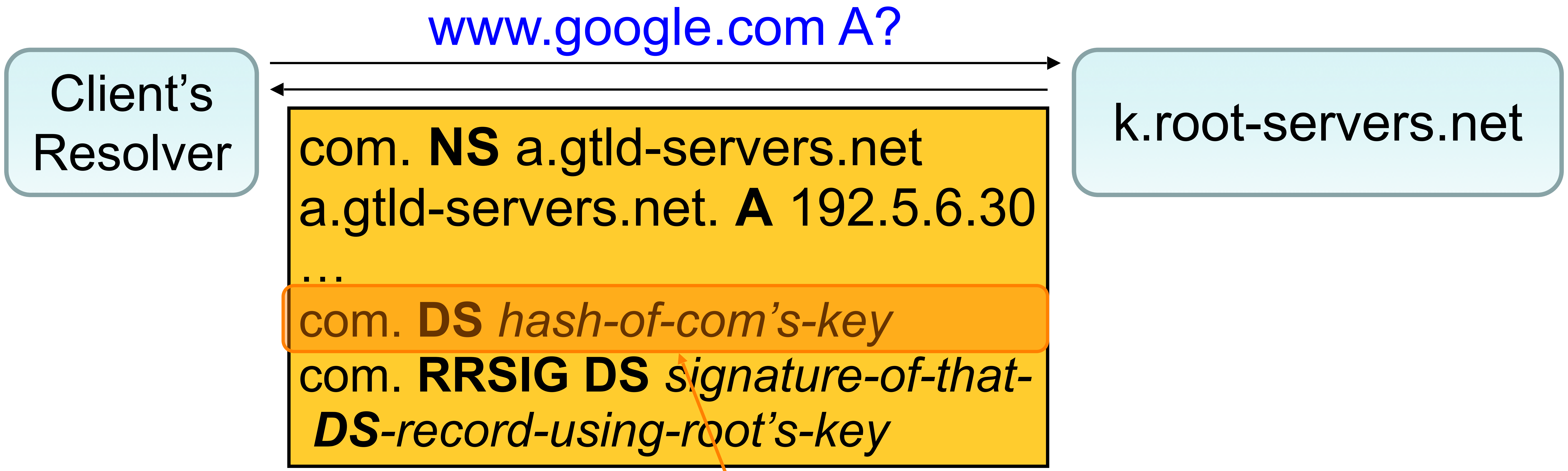
Client's Resolver

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net. **A** 192.5.6.30
…
com. **DS** *hash-of-com's-key*
com. **RRSIG DS** *signature-of-that-**DS**-record-using-root's-key*

Same as before …

DNSSEC (with simplifications):



New **RR** ("Delegation Signer") tells us if we have correct copy of `.com`'s public key (by comparing hash values)

DNSSEC (with simplifications):

Client's Resolver

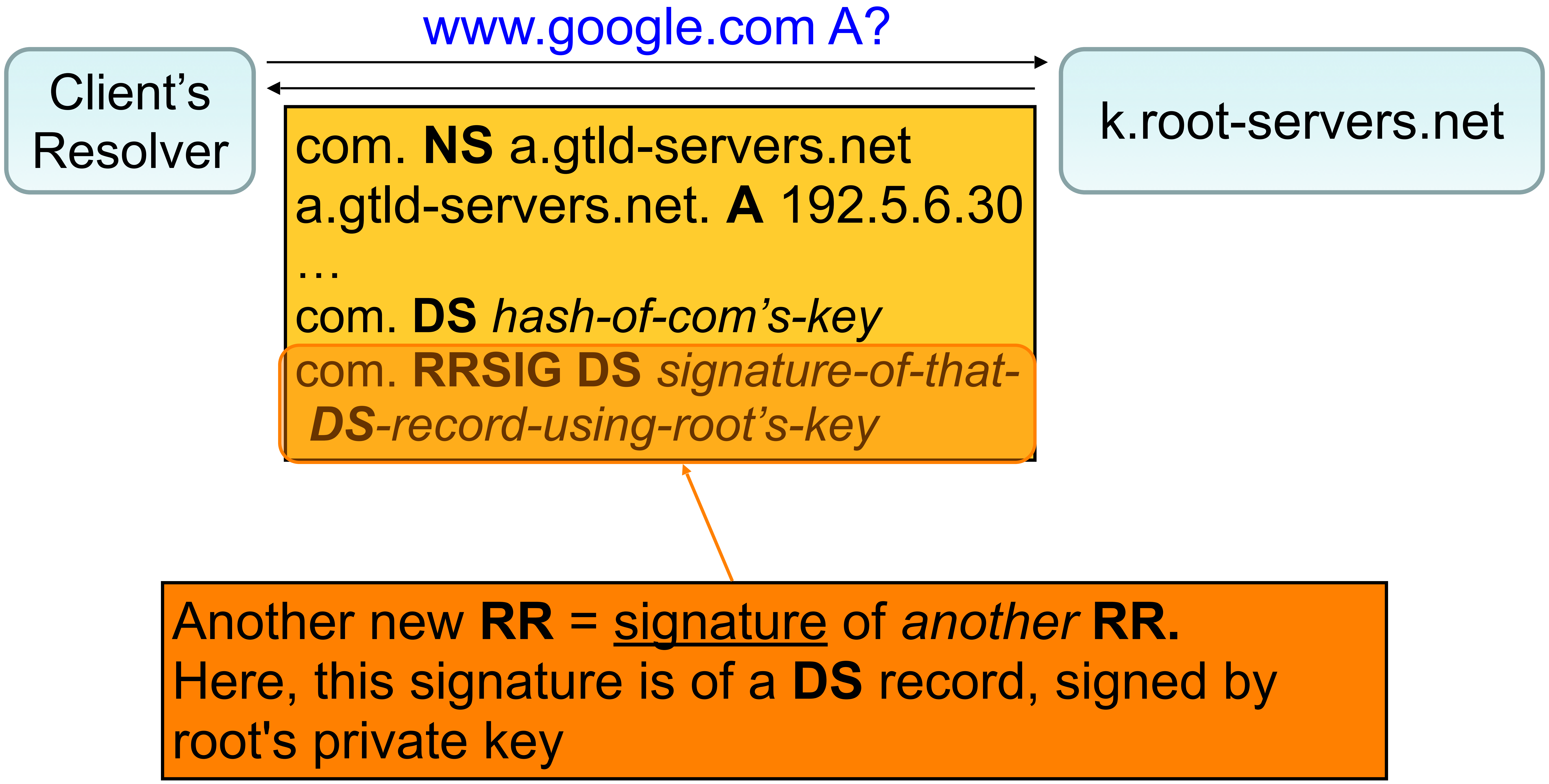
www.google.com A?

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net. **A** 192.5.6.30
…
com. **DS** *hash-of-com's-key*
com. **RRSIG DS** *signature-of-that-**DS**-record-using-root's-key*

Getting .com NS's key is a bit complicated...we'll talk about in a bit. Assume we can get it for now...

DNSSEC (with simplifications):



DNSSEC (with simplifications):

www.google.com A?

Client's Resolver

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net. **A** 192.5.6.30
…
com. **DS** *hash-of-com's-key*
com. **RRSIG DS** *signature-of-that-* ***DS****-record-using-root's-key*

Root's public key **hardwired** into resolvers. The client only proceeds with DNSSEC if it can validate the signature.

DNSSEC (with simplifications):

www.google.com A?

Client's Resolver

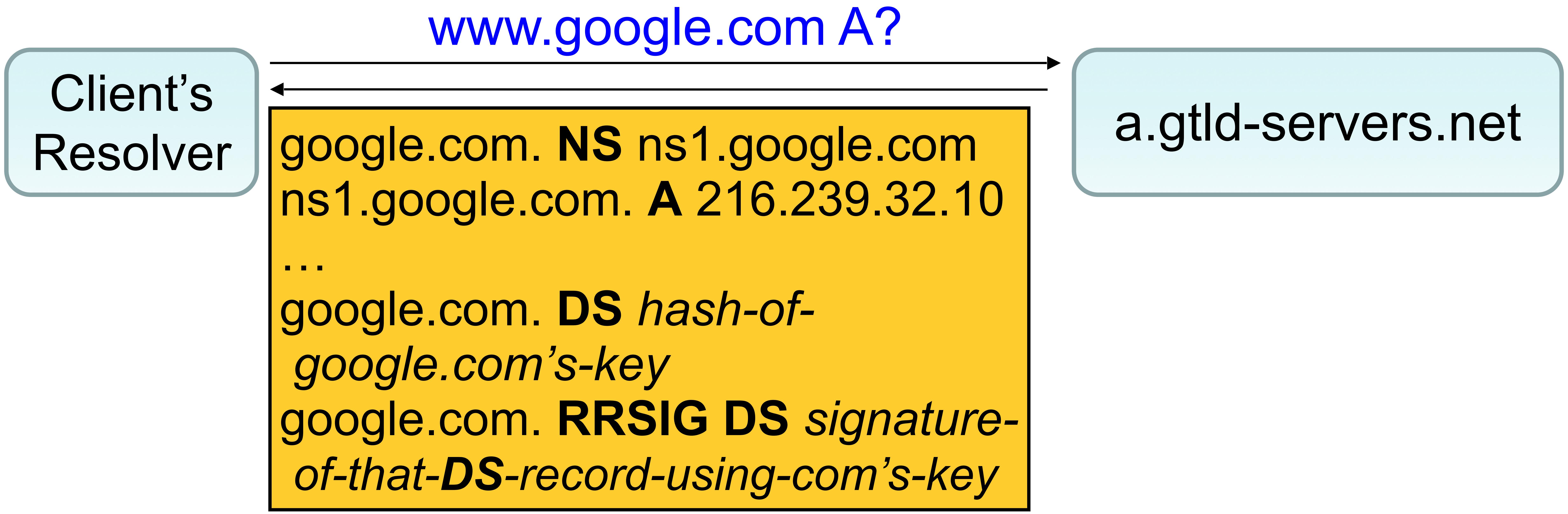
k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net. **A** 192.5.6.30
…
com. **DS** *hash-of-com's-key*
com. **RRSIG DS** *signature-of-that-*
***DS**-record-using-root's-key*

Note: there's no signature over the **NS** or **A** information! If DNS data tampered with, will find out later.

DNSSEC (with simplifications):

DNSSEC (with simplifications):



DNSSEC (with simplifications):

www.google.com A?

Client's Resolver

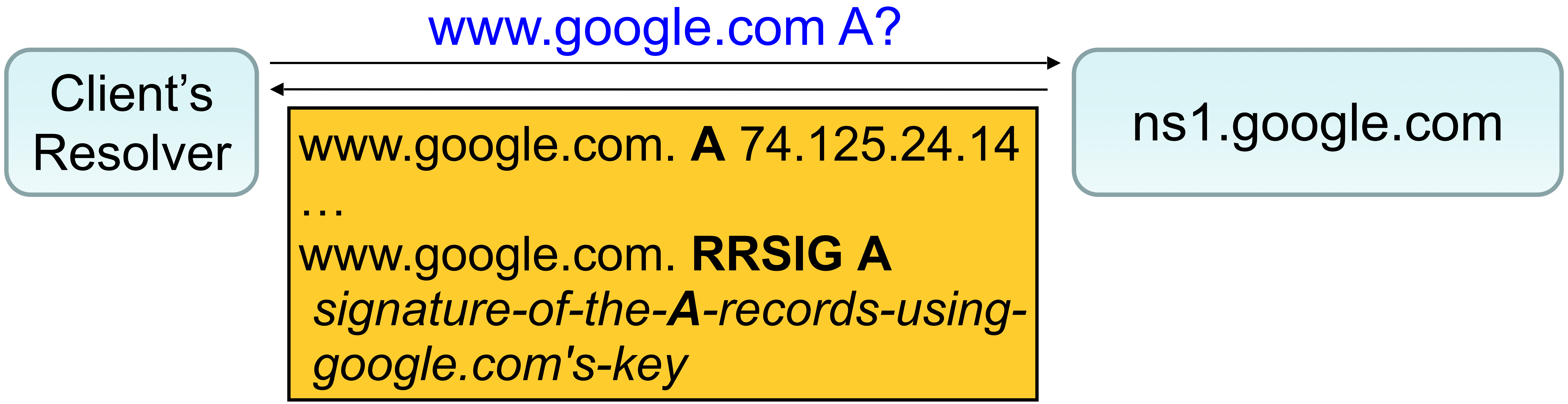
a.gtld-servers.net

google.com. **NS** ns1.google.com
ns1.google.com. **A** 216.239.32.10
…
google.com. **DS** *hash-of-google.com's-key*
google.com. **RRSIG DS** *signature-of-that-**DS**-record-using-com's-key*

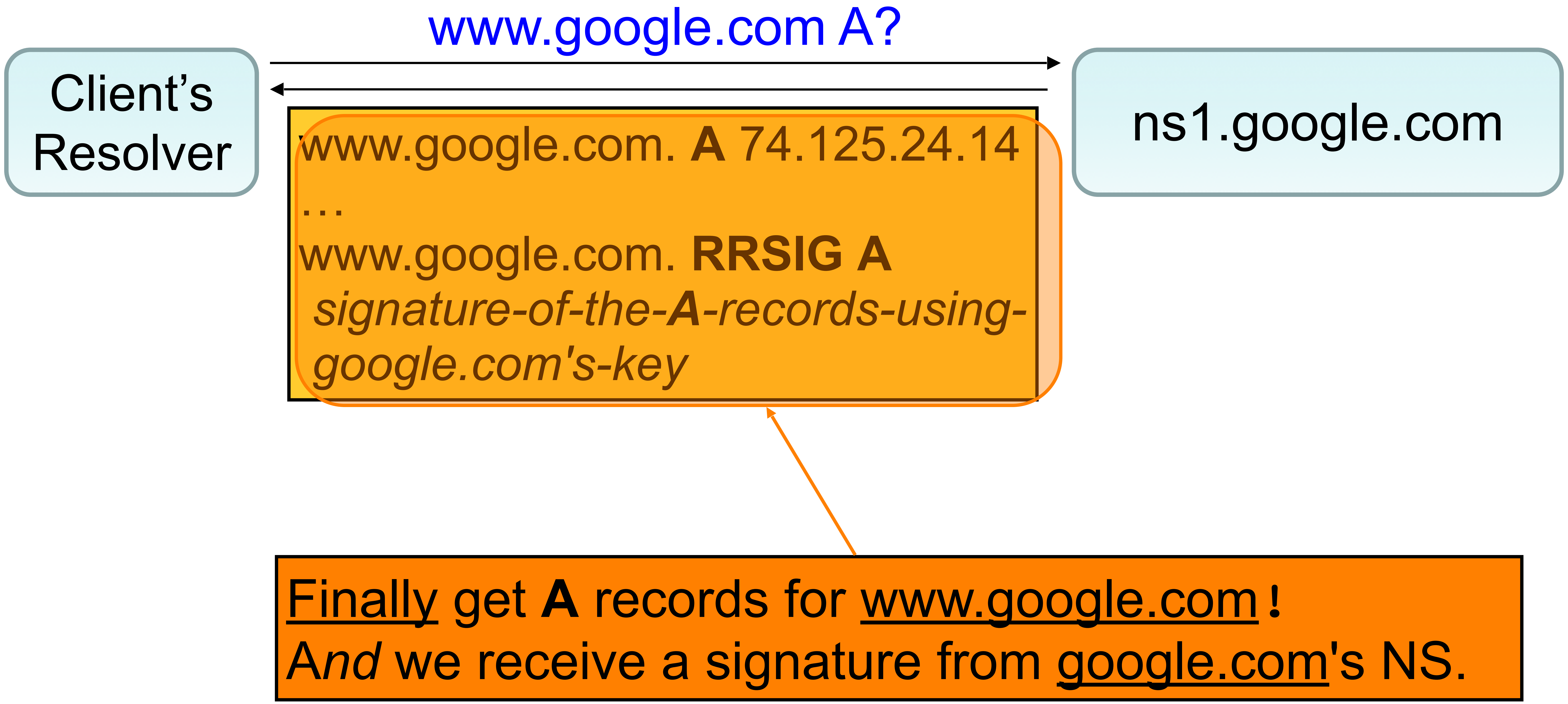
Similar as before:
- Identify `google.com`'s public key
- DS signed by `.com` NS's key

DNSSEC (with simplifications):

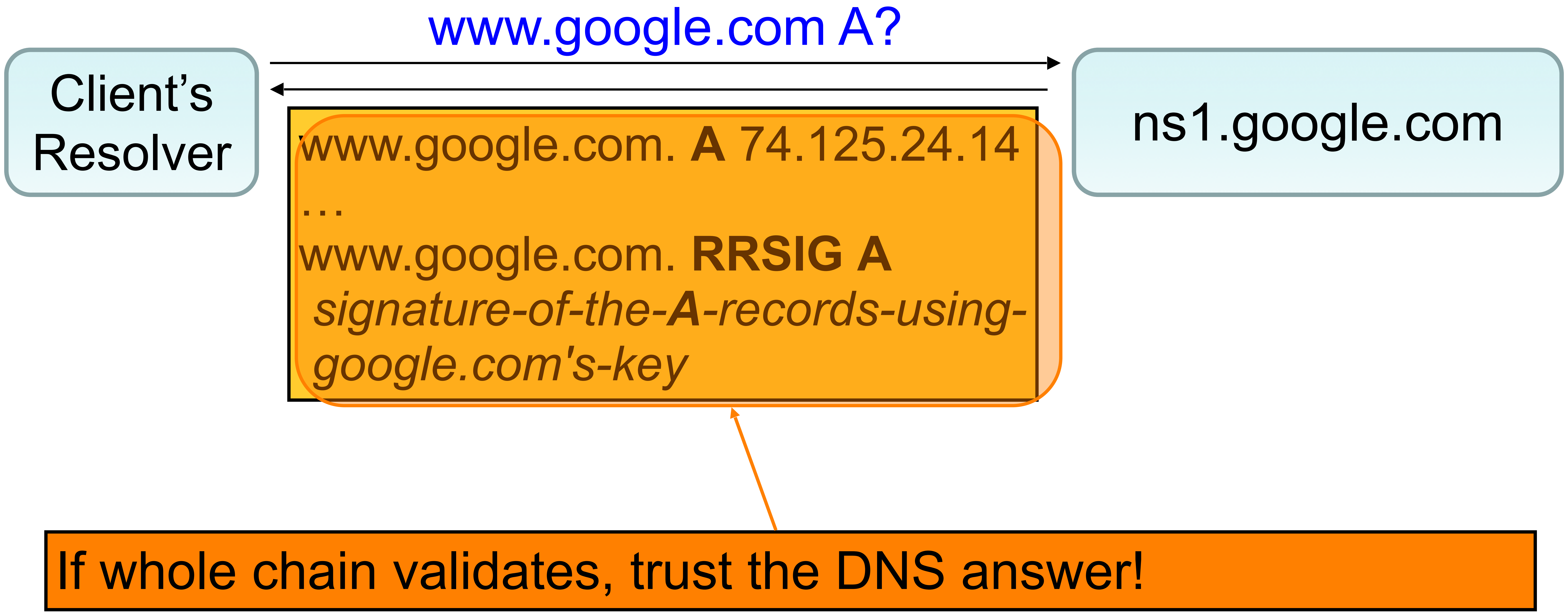
DNSSEC (with simplifications):



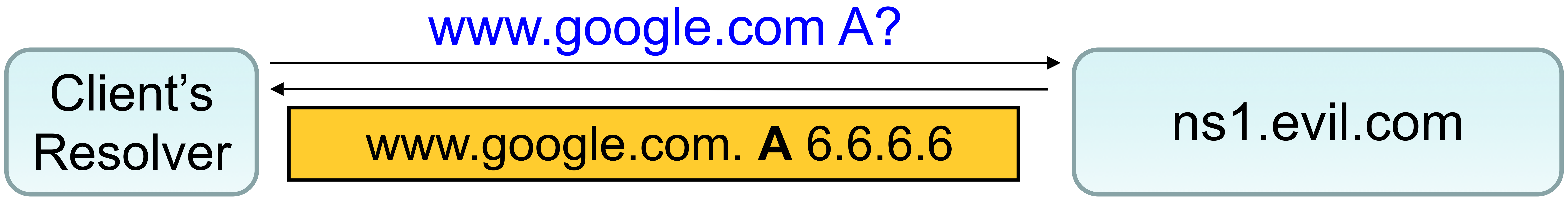
DNSSEC (with simplifications):



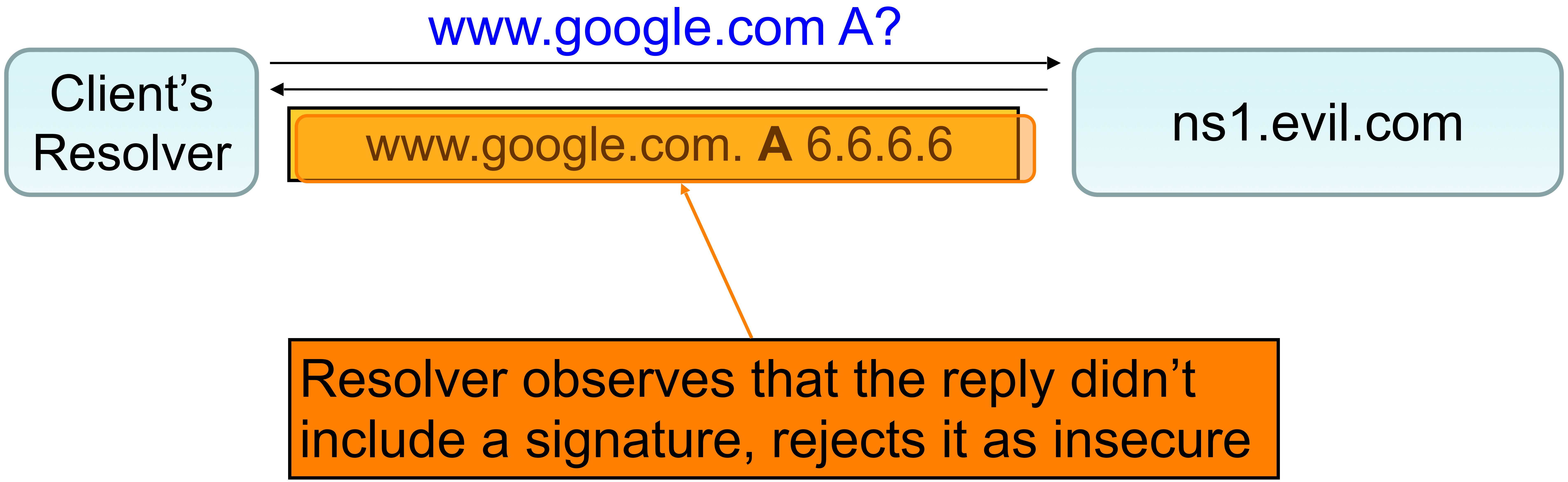
DNSSEC (with simplifications):



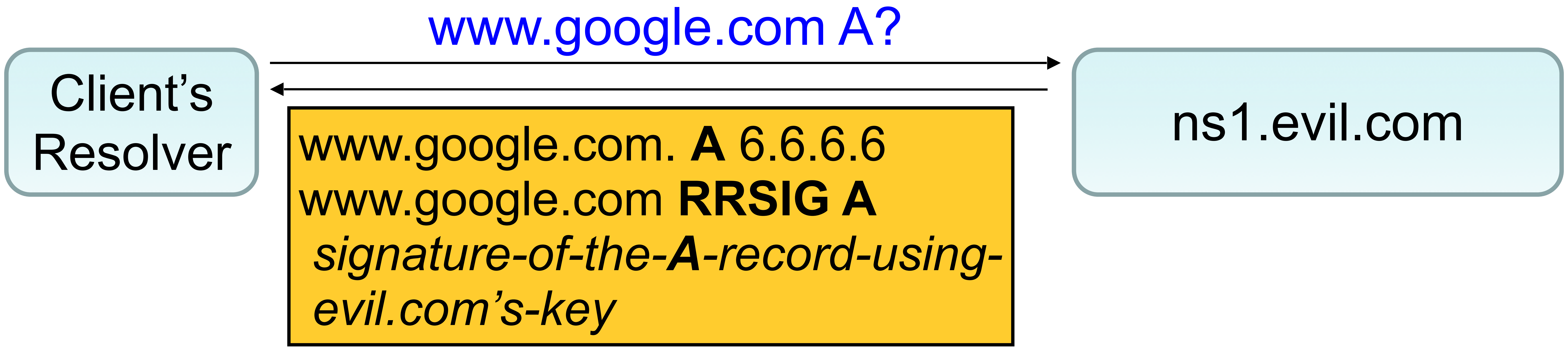
DNSSEC - Mallory attacks!



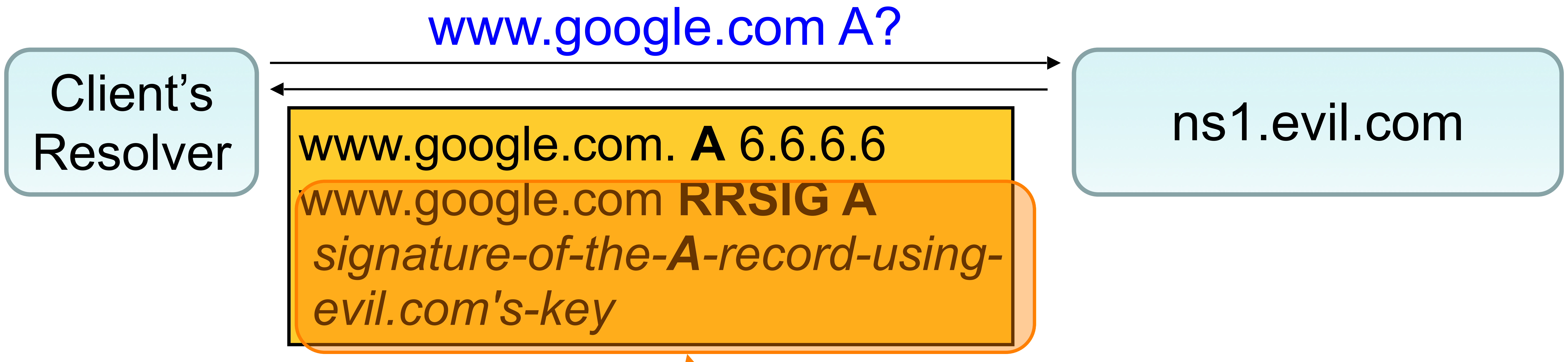
DNSSEC - Mallory attacks!



DNSSEC - Mallory attacks!

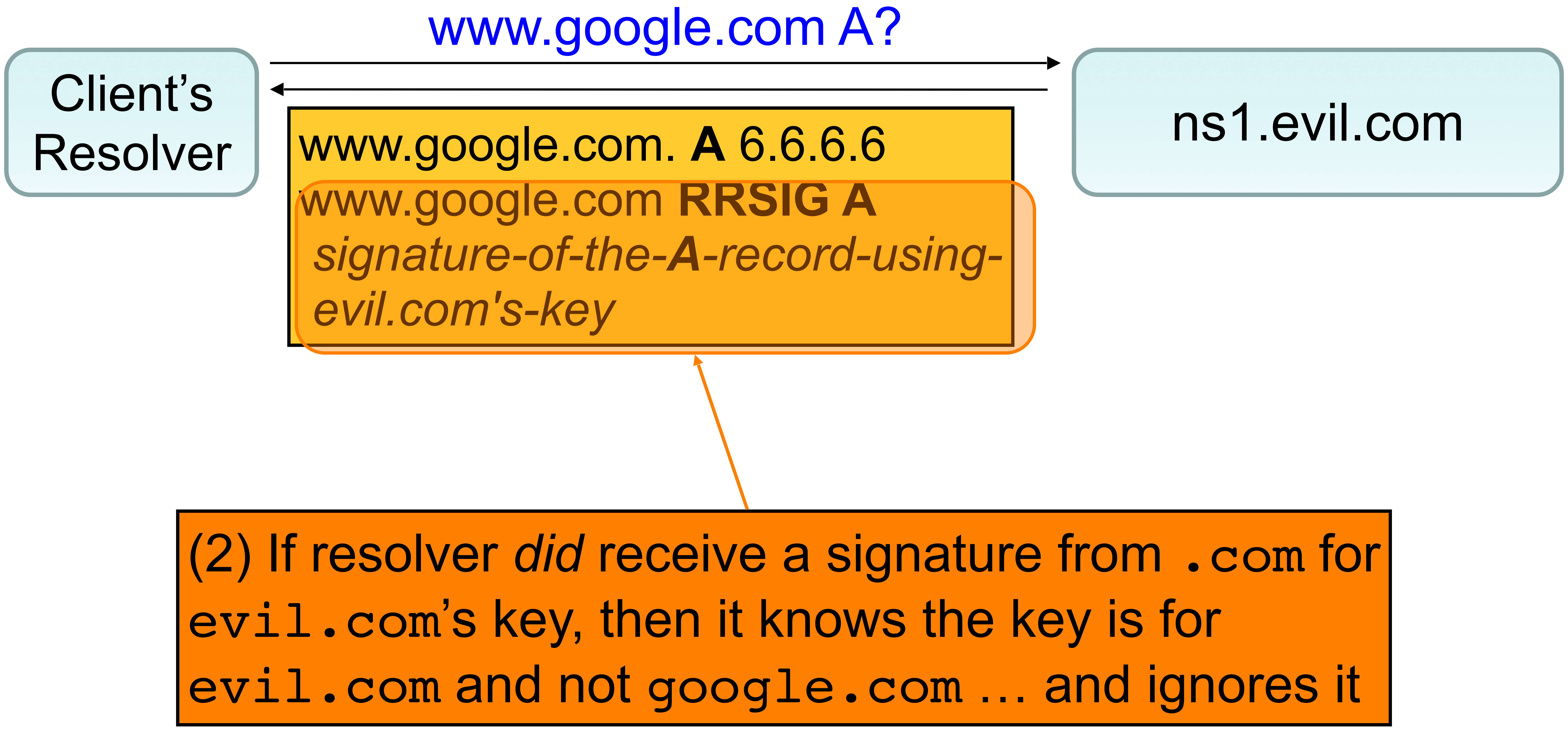


DNSSEC - Mallory attacks!

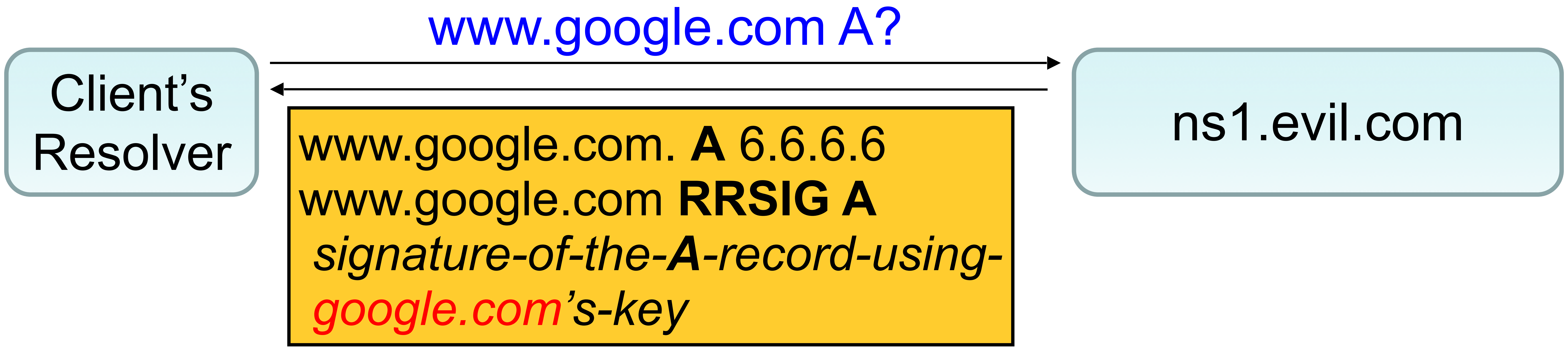


(1) If resolver didn't receive a signature from `.com` for `evil.com`'s key, then it can't validate this signature & ignores reply since it's not properly signed …

DNSSEC - Mallory attacks!



DNSSEC - Mallory attacks!



DNSSEC - Mallory attacks!

www.google.com A?

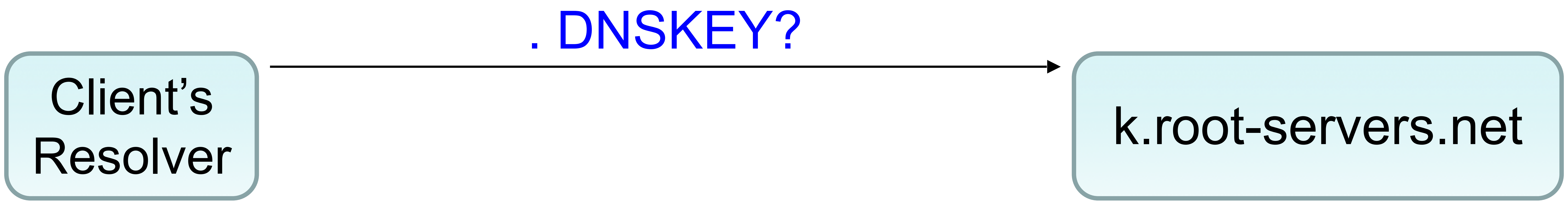
Client's Resolver

ns1.evil.com

www.google.com. **A** 6.6.6.6
www.google.com **RRSIG A**
*signature-of-the-**A**-record-using-google.com's-key*

If signature **actually** comes from `google.com`'s key, resolver will believe it …
… but no such signature should exist unless either:
(1) `google.com` *intended* to sign the RR, or
(2) `google.com`'s private key was compromised

DNSSEC: Accessing keys



To build up the keys needed for validation, our client contacts each name server in the DNS hierarchy asking it for all of its associated keys.

Here we ask the root for its keys (one of which we already know as our **trust anchor**).

DNSSEC: Accessing keys

Client's Resolver

. DNSKEY?

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

DNSSEC: Accessing keys

. DNSKEY?

Client's Resolver

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

Each **DNSKEY** is a public key plus a description of the algorithms it's associated with (e.g., RSA+SHA256)

DNSSEC: Accessing keys

. DNSKEY?

Client's Resolver

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

The **KSK** is used to sign all of the **DNSKEY** entries in the zone.

DNSSEC: Accessing keys

. DNSKEY?

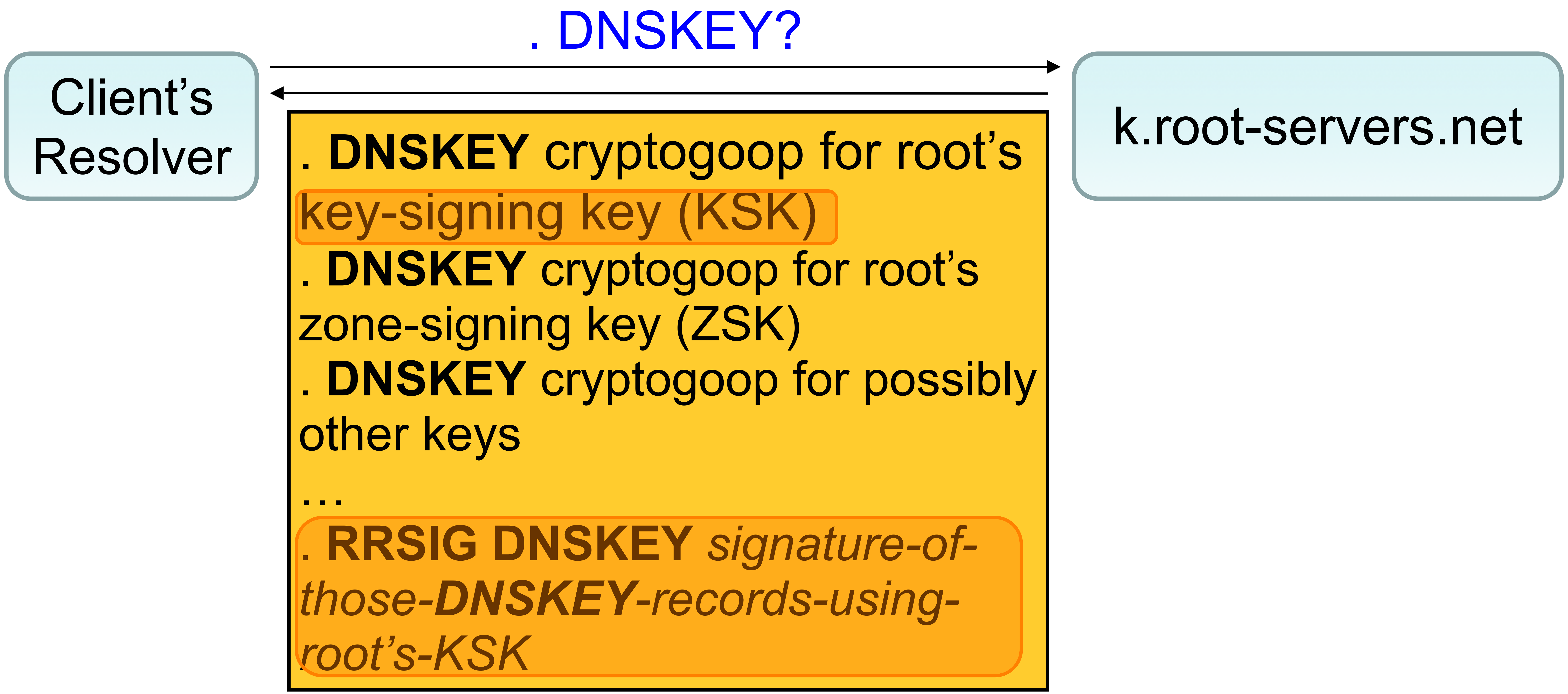
Client's Resolver

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

The client has a hash of the root's **KSK** hardwired into its config as a trust anchor.

DNSSEC: Accessing keys



For everything below the root (e.g., `.com` and `google.com`) we get a hash of the KSK via a **DS** record, as shown earlier, so we can tell if we get the right KSK in a **DNSKEY** entry.

DNSSEC: Accessing keys

. DNSKEY?

Client's Resolver

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

The **ZSK** is used for signing all of the other RRSIG entries in the zone, including DS records for subzones.
(E.g., `.com` signs its **DS** record for `google.com` using `.com`'s **ZSK**

DNSSEC: Accessing keys

. DNSKEY?

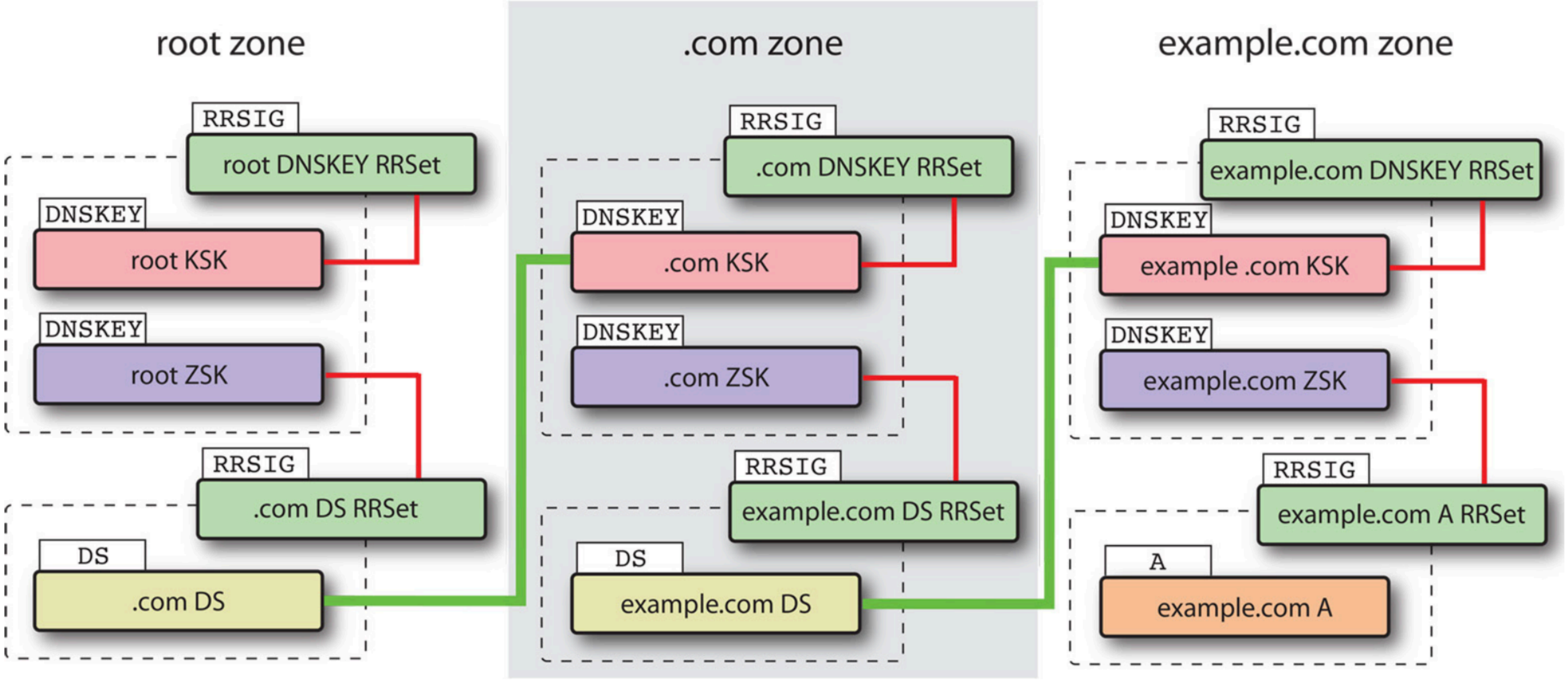
Client's Resolver

k.root-servers.net

. **DNSKEY** cryptogoop for root's key-signing key (KSK)
. **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
. **DNSKEY** cryptogoop for possibly other keys
…
. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

Having separate key-signing-keys vs. zone-signing-keys allows a zone to change its **ZSK** without needing to get its parent to re-sign, since parent only signs the **KSK**. Enables frequent *key rollover*.

End-to-end look



Issues With DNSSEC ?

- Issue #1: Replies are Big
  - DoS amplification
  - Increased latency on low-capacity links
  - Headaches w/ older libraries that assume replies < 512B

Issues With DNSSEC ?

- Issue #1: Replies are Big
  - DoS amplification
  - Increased latency on low-capacity links
  - Headaches w/ older libraries that assume replies < 512B

- Issue #2: *Partial deployment*
  - What do you do with unsigned/unvalidated results?
  - If you trust them, weakens incentive to upgrade
  - If you don't trust them, a whole lot of things break

Issues With DNSSEC, con't

- Issue #3: *Management headaches*
  - What happens if when updating your site's keys you make a mistake?
  - Suddenly your Entire Site Breaks
- Issue #4: Negative results ("no such name")
  - What statement does the nameserver sign?
  - If "`gabluph.google.com`" doesn't exist, then have to do dynamic key-signing (expensive) for any bogus request
    - DoS vulnerability
  - Instead, sign (off-line) statements about order of names
    - E.g., sign "`gabby.google.com` followed by `gabrunk.google.com`"
    - Thus, can see that `gabluph.google.com` can't exist
  - But: now attacker can enumerate all names that exist :-(

Issues With DNSSEC, con't

- Issue #5: *Who do you really trust?*
  - For your laptop (say), who does all the "grunt work" of fetching keys & validating DNSSEC signatures?
- Convenient answer: your laptop's local resolver
  - … which you acquire via DHCP in your local coffeeshop
  - I.e., exactly the most-feared potentially untrustworthy part of the DNS resolution process!
- Alternatives?

  ⇒ Your laptop needs to do all the validation work itself :-(

Summary of DNSSEC

- DNSSEC: provides object security for DNS results
  - Just integrity & authentication, not confidentiality
  - No client/server setup "dialog"
  - Tailored to be caching-friendly
  - Underlying security dependent on trust in Root Name Server's key …
  - … plus support provided by every level of DNS hierarchy from Root to final name server… and local resolver!