

# Lecture 21: DNSSEC

<https://cs161.org>

# Announcements

- Discussion sections are online
- Office hours are available online: [oh.cs161.org](http://oh.cs161.org)
- I'd love to take your questions and feedback during lecture in Zoom's text chat

# DNSSEC

- DNSSEC = standardized DNS security extensions currently being deployed
- Aims to ensure **integrity** of the DNS lookup results (to ensure correctness of returned IP addresses for a domain name)

Q: what attack is it trying to prevent?

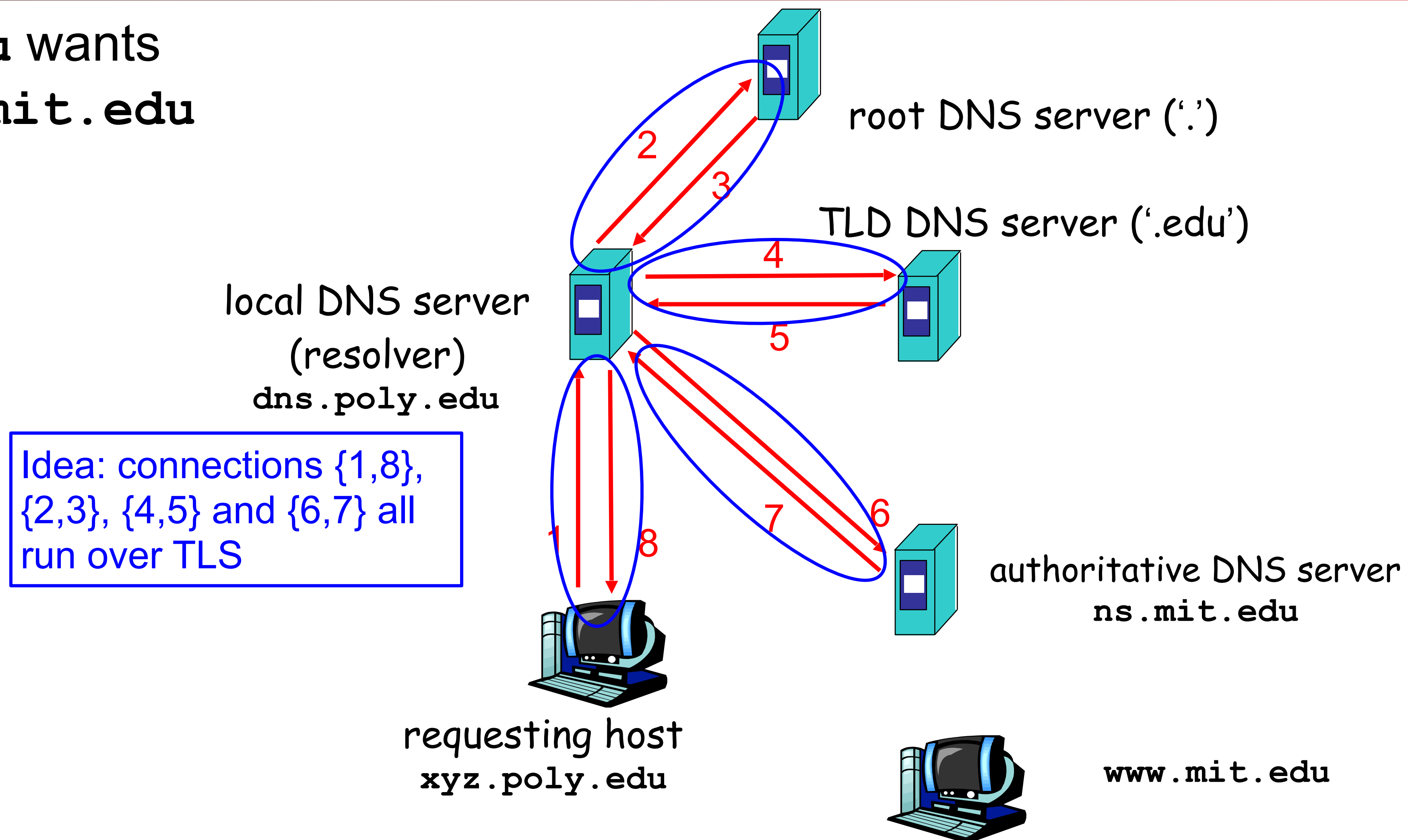
A: attacker changes DNS record result with an incorrect IP address for a domain

# Securing DNS Lookups

- How can we ensure that when clients look up names with DNS, they can **trust** the answers they receive?
- Idea #1: do DNS lookups over TLS (SSL)
- Background: TLS is a protocol for building an encrypted connection, using public-key exchange to exchange a session key, then using encryption and a message authentication code on all data sent over the connection

# Securing DNS Using TLS

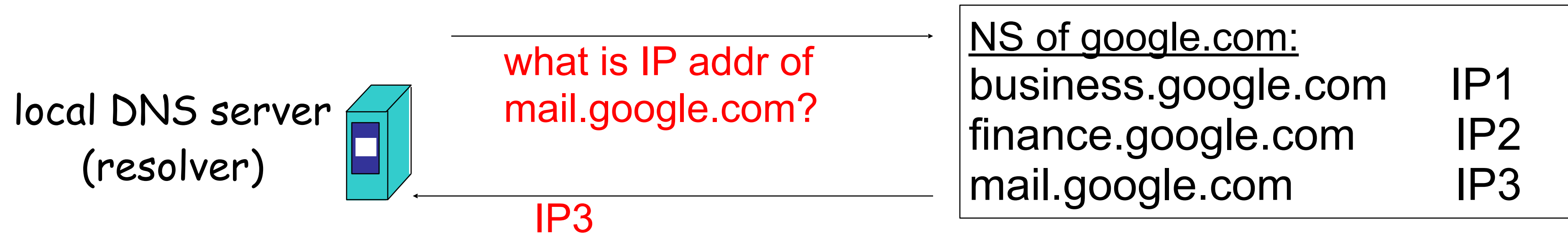
Host at `xyz.poly.edu` wants  
IP address for `www.mit.edu`



# Securing DNS Lookups

- How can we ensure that when clients look up names with DNS, they can trust the answers they receive?
- Idea #1: do DNS lookups over TLS
  - **Performance**: DNS is very lightweight. TLS is not.
  - **Caching**: crucial for DNS scaling. But then how do we keep authentication assurances?
  - **Security**: must trust the resolver.  
*Object security vs. Channel security*  
How do we know which name servers to trust?
- Idea #2: make DNS results like *certs*
  - I.e., a **verifiable signature** that guarantees who generated a piece of data; signing happens **off-line**

# Scratchpad – let's design it together



Q: How can we ensure returned result is correct?

A: Have google.com NS sign IP3

Q: What should the signature contain?

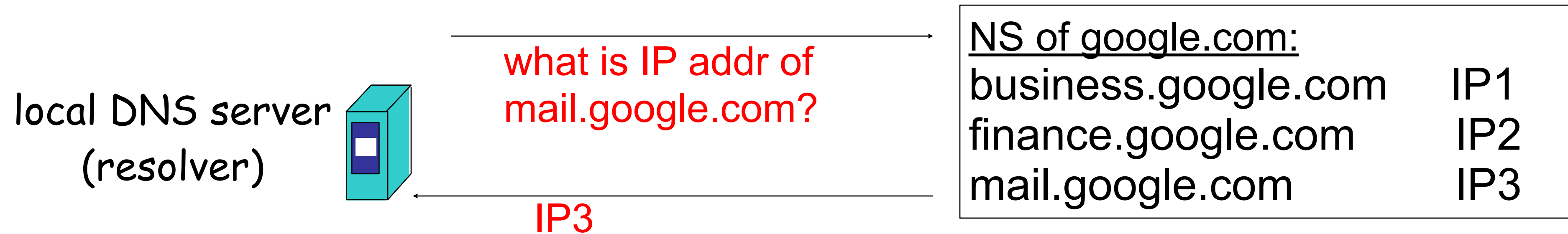
A: At least the domain name, IP address, cache time

Q: How do we know google.com's PK?

A: The .com NS can give us a certificate on it



# Scratchpad – let's design it together



Q: How do we know .com's PK?

A: Chain of certificates, like for the web, rooted in the PK of the root name server

Q: How do we know the PK of the root NS?

A: Hardcoded in the resolvers

Q: How does the resolver verify a chain of certificates?



# Scratchpad – let's design it together



Q: How can we ensure returned result is correct?

A: Have google.com NS sign the “no record” response  
sign(“dog.google.com” does not exist)

But it is expensive to sign online.

Q: What problem can this cause?

A: DoS due to an amplification of effort between query and response.

# Scratchpad – let's design it together



Q: How can we sign the no-record response offline?

A: We don't know which are all the non-existent domains we might be asked for, but we can sign consecutive domains that do exist.

`sign(["business.google.com", "finance.google.com"])`

This indicates absence of a name in the middle and is cacheable.

Q: What problem can this cause?

A: Enumeration attack. An attacker can issue queries for things that do not exist and obtains intervals of all the things that exist until it mapped the whole space.

# DNSSEC

Now let's go through it slowly...

# DNSSEC

- Key idea:
  - Sign all DNS records. Signatures let you verify answer to DNS query, without having to trust the network or resolvers involved.
- Remaining challenges:
  - DNS records change over time
  - Distributed database: No single central source of truth

# Operation of DNSSEC

- 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:

www.google.com A?

Client's  
Resolver

k.root-servers.net

# Ordinary DNS:

www.google.com A?

Client's  
Resolver

k.root-servers.net

We start off by sending the query to one of the root name servers. These range from a.root-servers.net through m.root-servers.net. Here we just picked one.



# Ordinary DNS:

www.google.com A?

Client's  
Resolver

```
com. NS a.gtld-servers.net  
a.gtld-servers.net A 192.5.6.30  
...
```

k.root-servers.net

# Ordinary DNS:

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  
...
```

The reply *didn't include an answer* for `www.google.com`. That means that `k.root-servers.net` is instead telling us *where to ask next*, namely one of the name servers for `.com` specified in an **NS** record.

# Ordinary DNS:

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  
...
```

This *Resource Record (RR)* tells us that one of the name servers for .com is the host a.gtld-servers.net. (GTLD = Global Top Level Domain.)

# Ordinary DNS:

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  
...
```

This **RR** tells us that an Internet address (“**A**” record) for `a.gtld-servers.net` is `192.5.6.30`. That allows us to know where to send our next query.

# Ordinary DNS:

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  
...
```

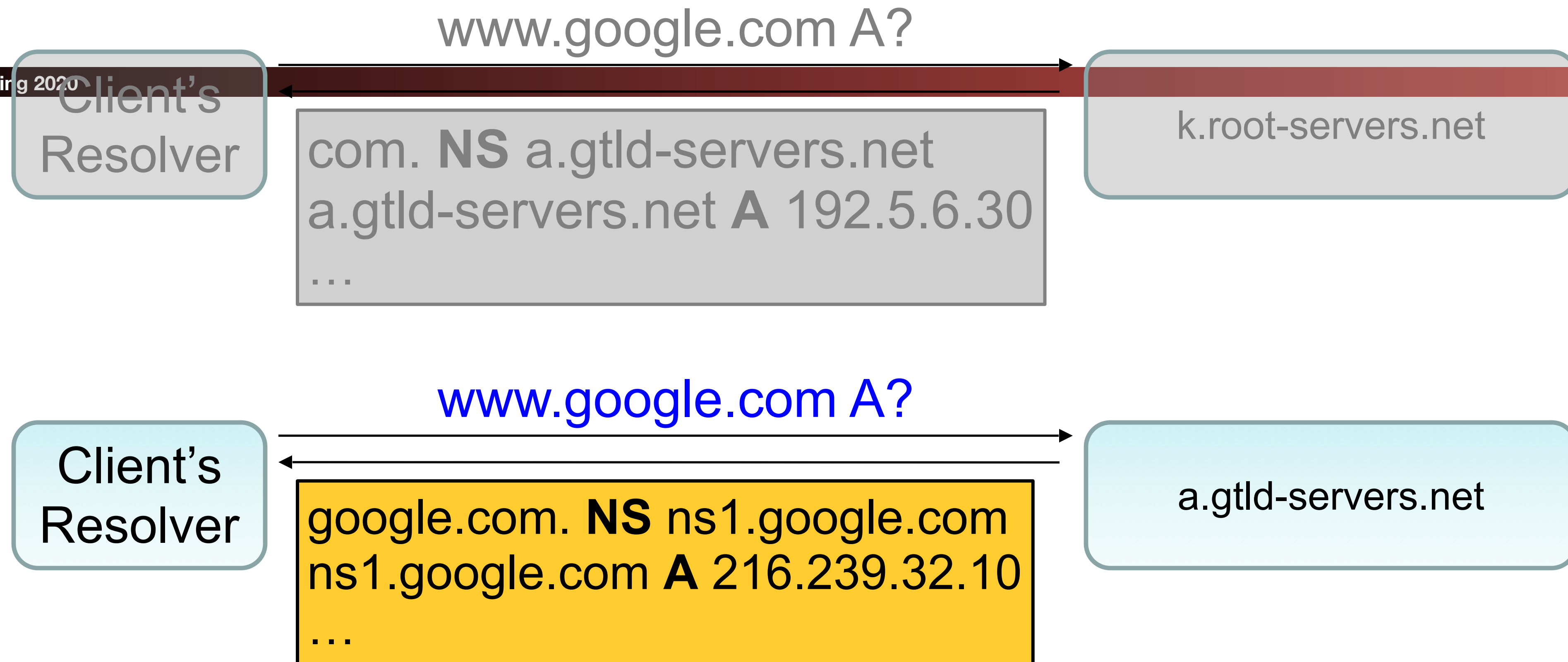
The actual response includes a bunch of **NS** and **A** records for additional .com name servers, which we omit here for simplicity.

# Ordinary DNS:



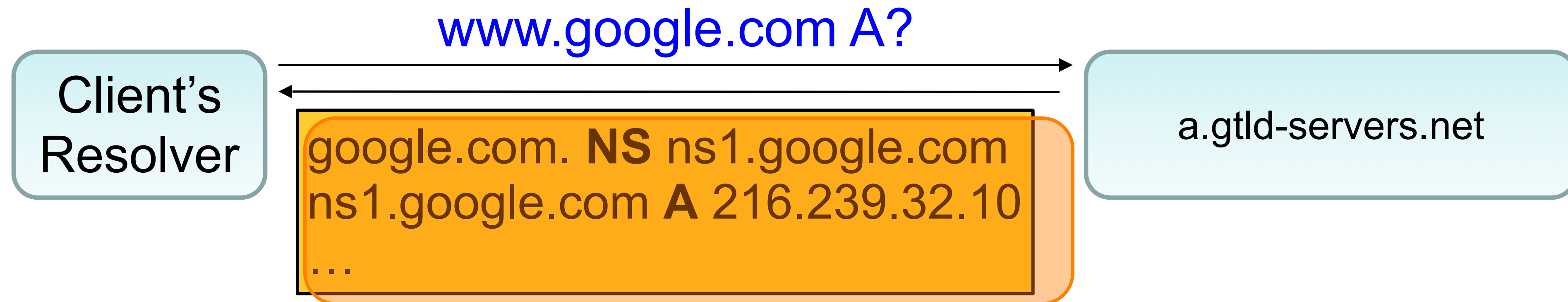
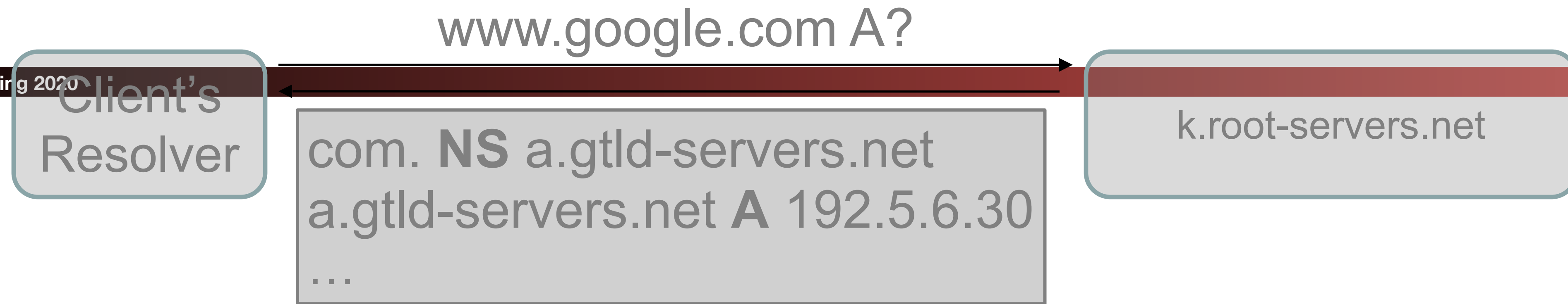
We send the same query to one of the .com name servers we've been told about

# Ordinary DNS:



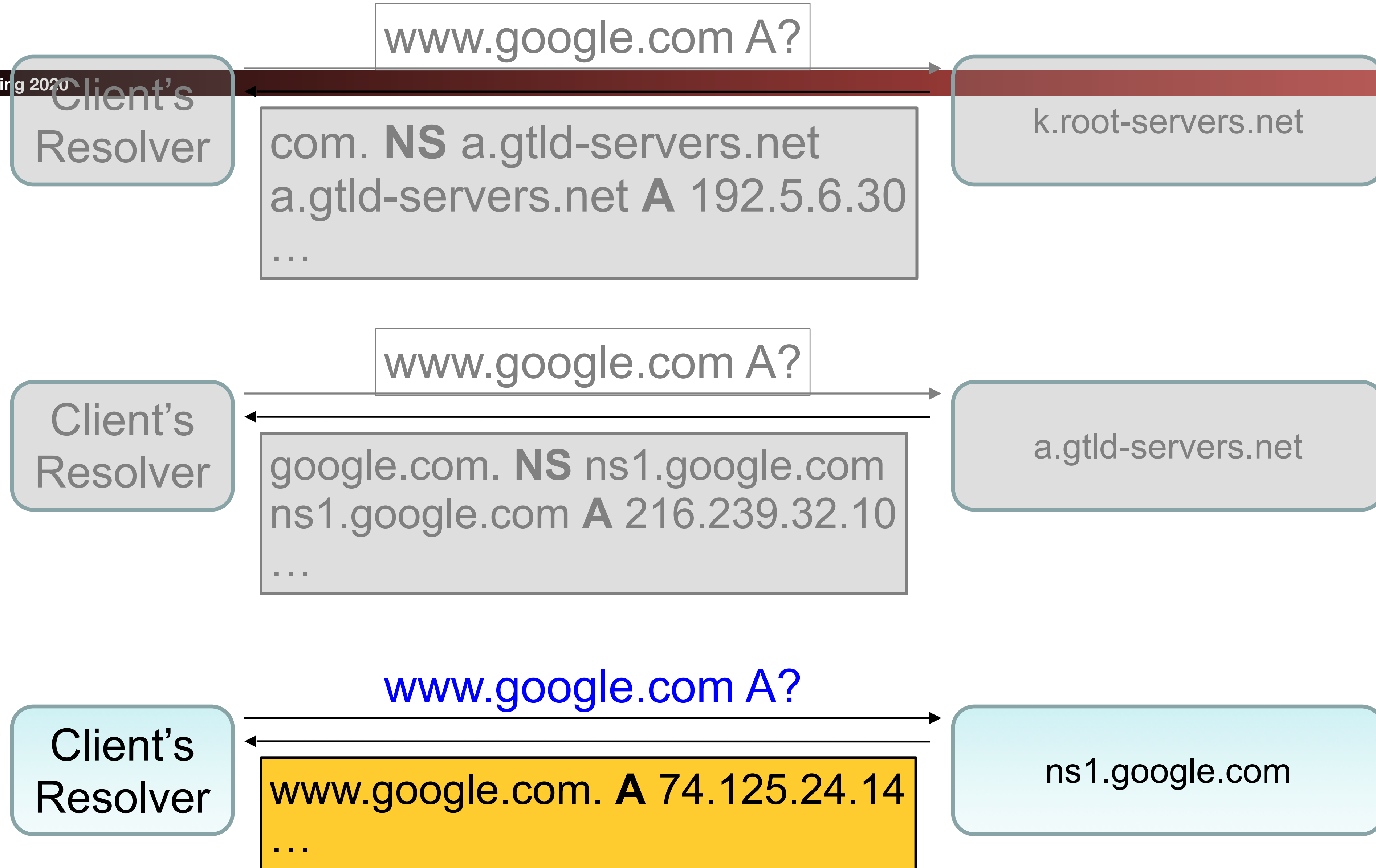


# Ordinary DNS:

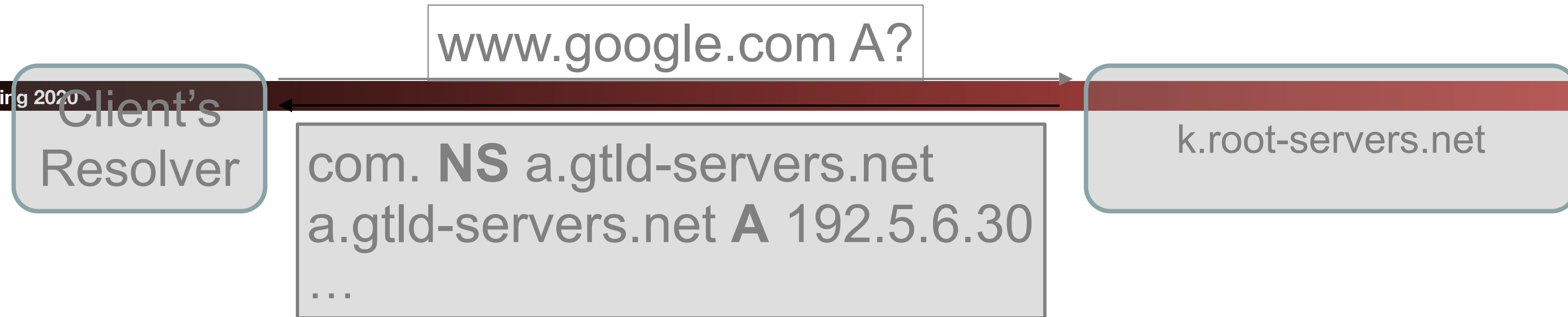


That server again doesn't have a direct answer for us, but tells us about a google.com name server we can try

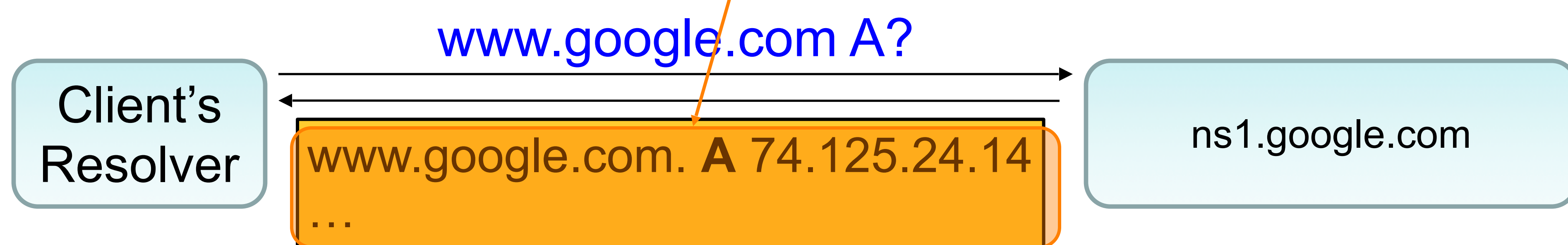
# Ordinary DNS:



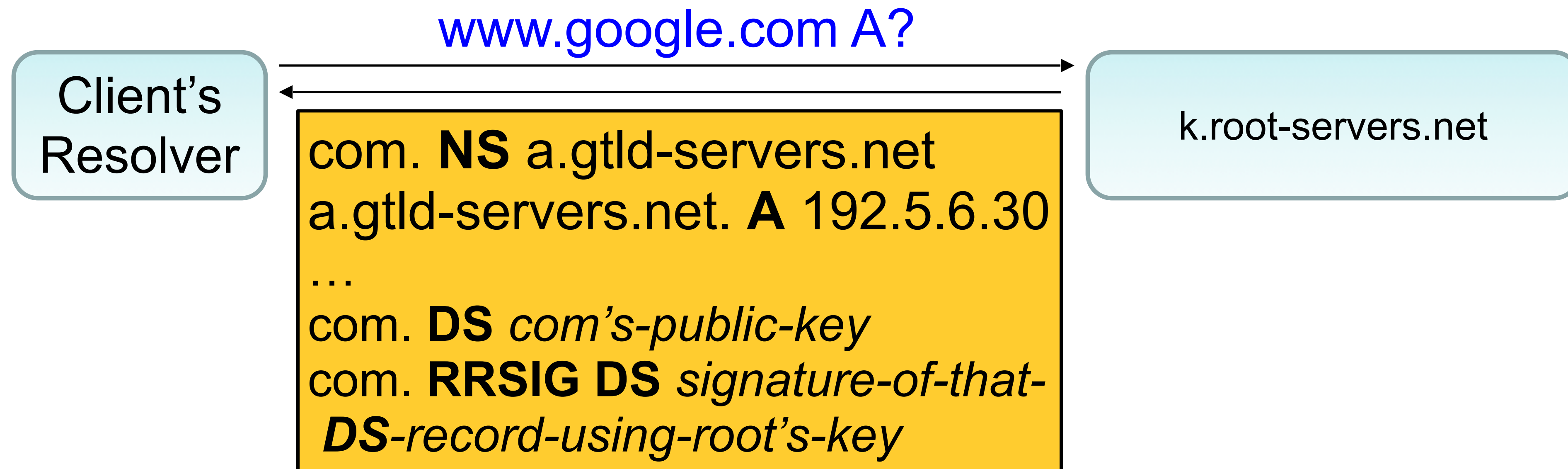
# Ordinary DNS:



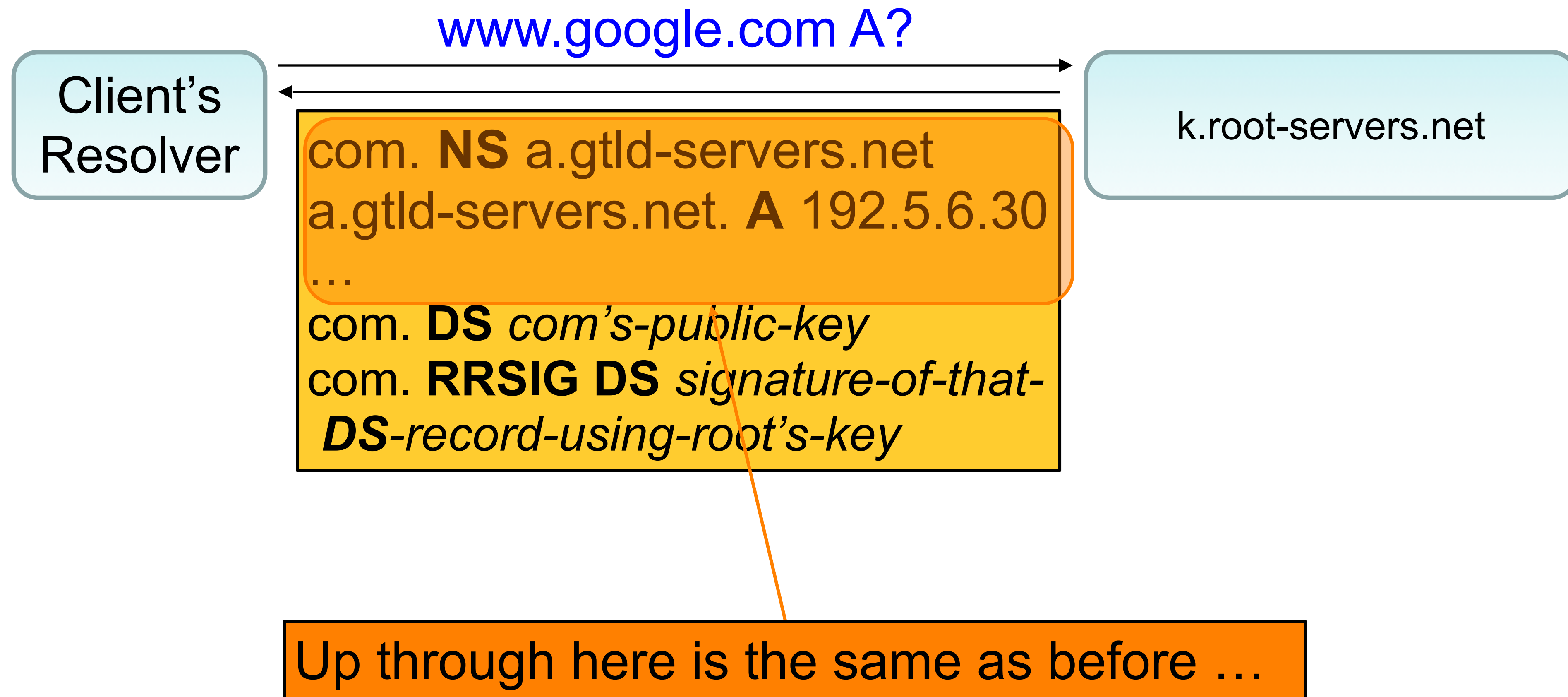
Trying one of the google.com name servers then gets us an answer to our query, and we're good-to-go ...  
... though with **no confidence** that an attacker hasn't led us astray with a bogus reply somewhere along the way :-)



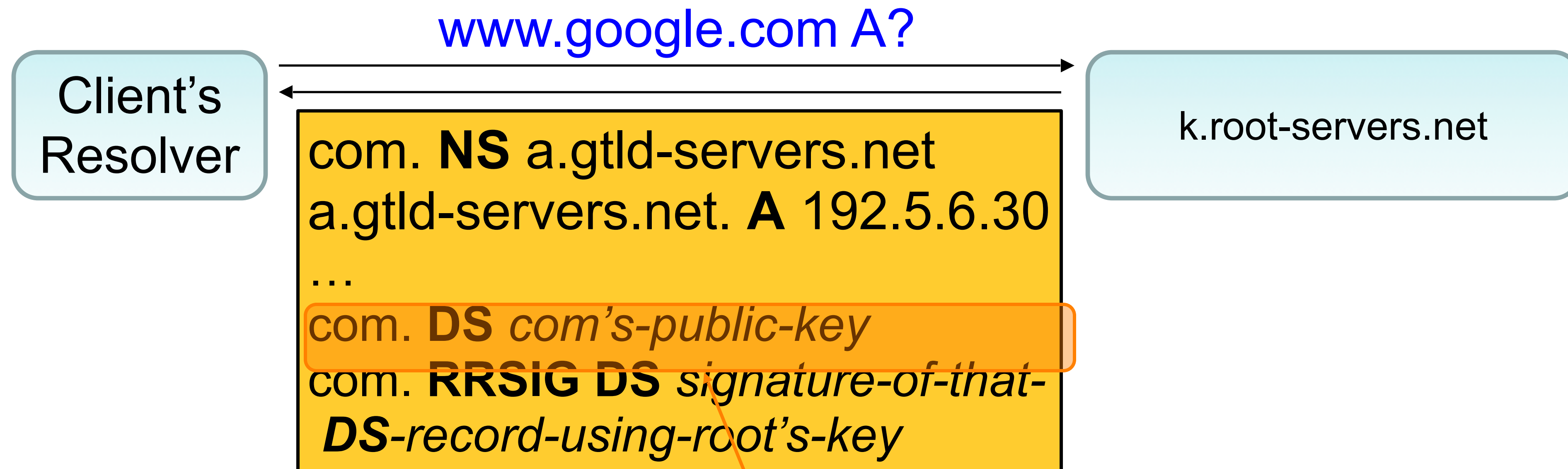
# DNSSEC (with simplifications):



# DNSSEC (with simplifications):

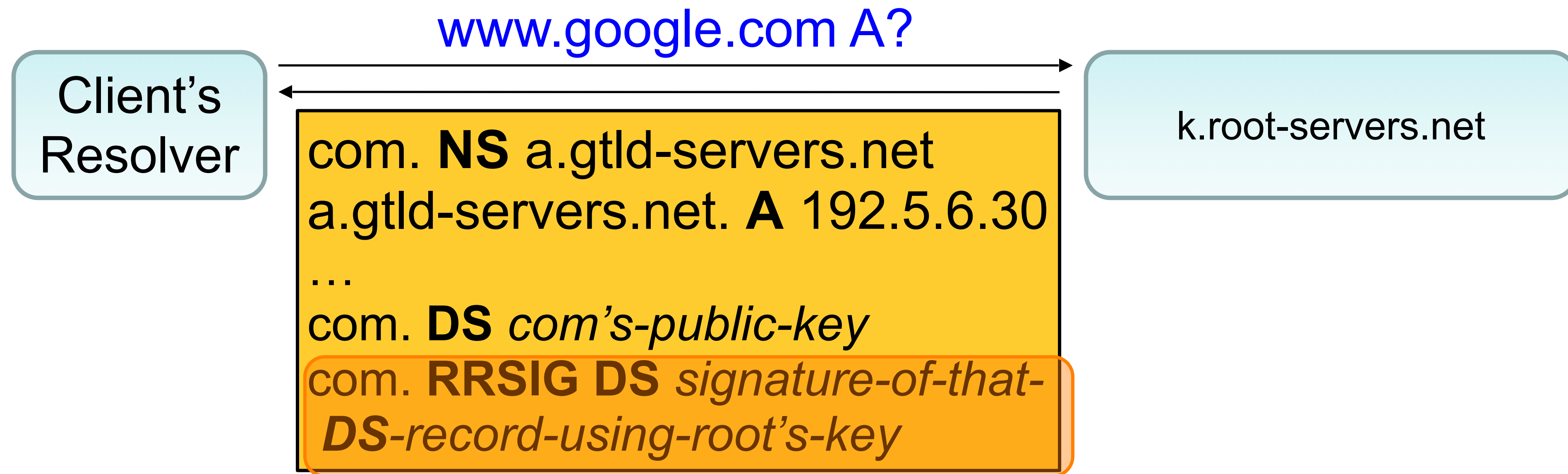


# DNSSEC (with simplifications):



This new **RR** (“Delegation Signer”) lists .com’s public key

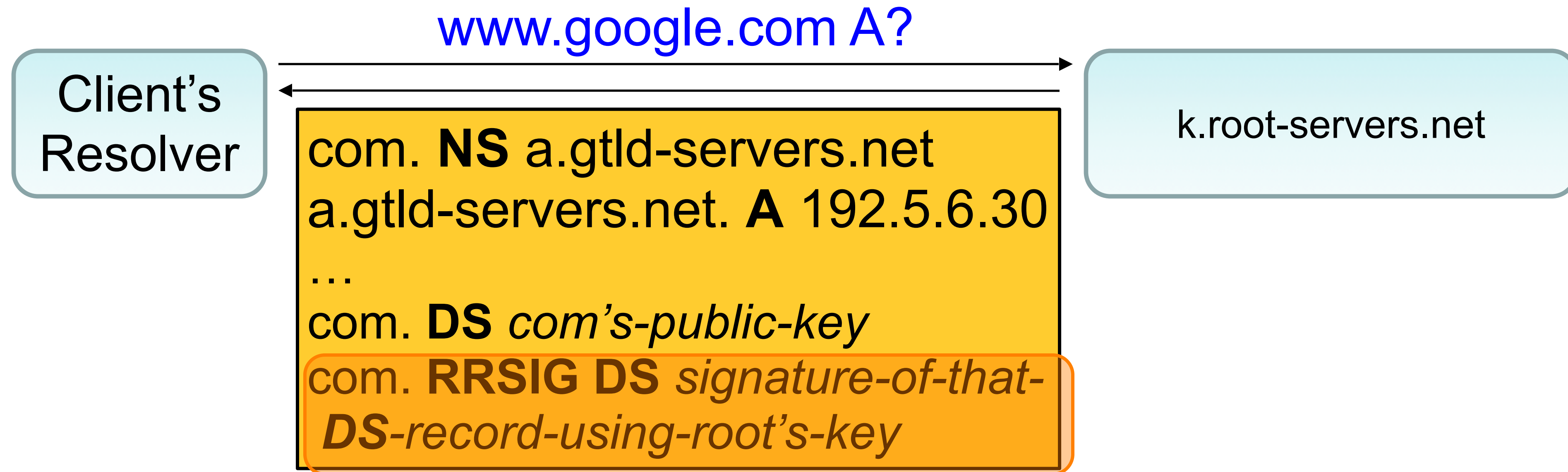
# DNSSEC (with simplifications):



This new **RR** specifies a signature (**RRSIG**) over *another RR* ... in this case, the signature covers the above **DS** record, and is made using the root's private key



# DNSSEC (with simplifications):



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

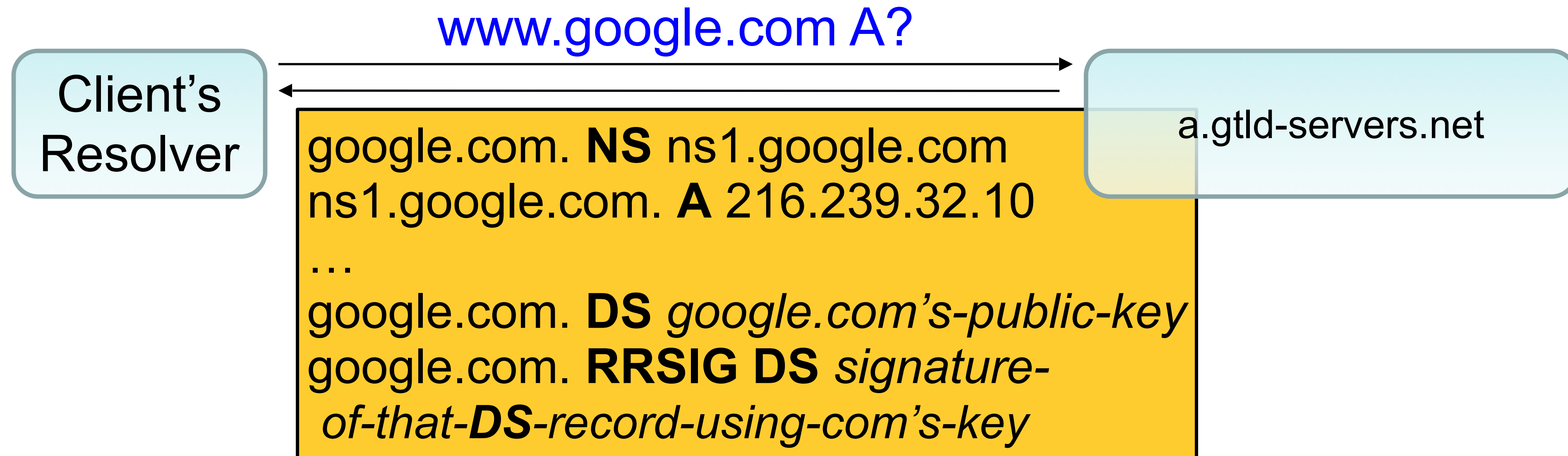
# DNSSEC (with simplifications):



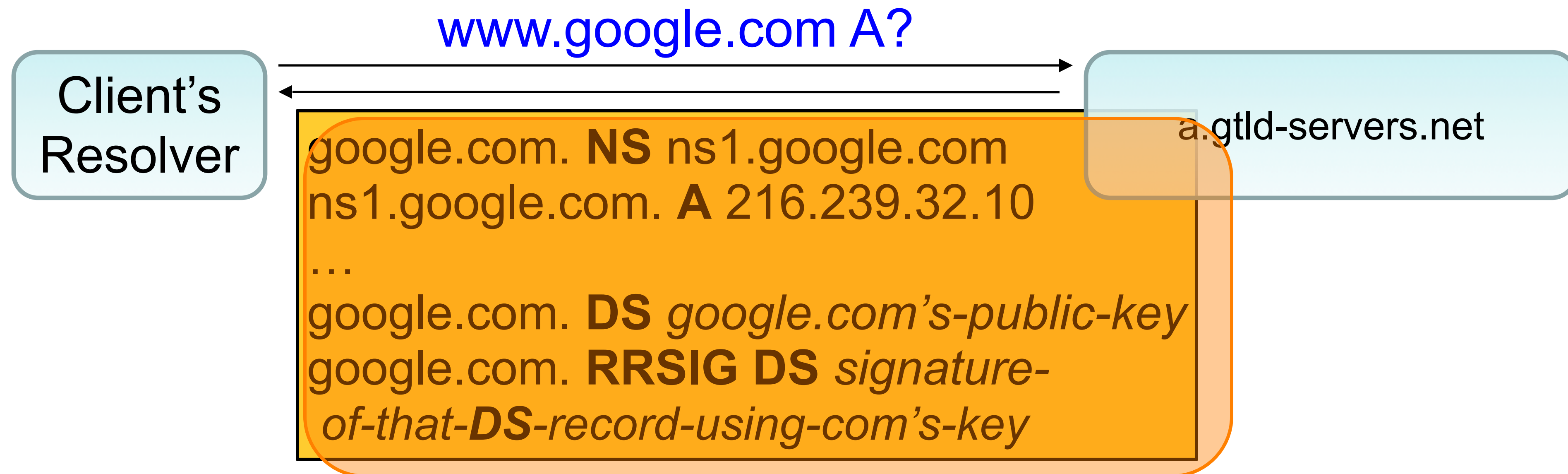
The resolver again proceeds to trying one of the name servers it's learned about.

Nothing guarantees this is a legitimate name server for the query!

# DNSSEC (with simplifications):



# DNSSEC (with simplifications):



Back comes similar information as before: google.com's public key, signed by .com's key (which the resolver trusts because the root signed information about it)

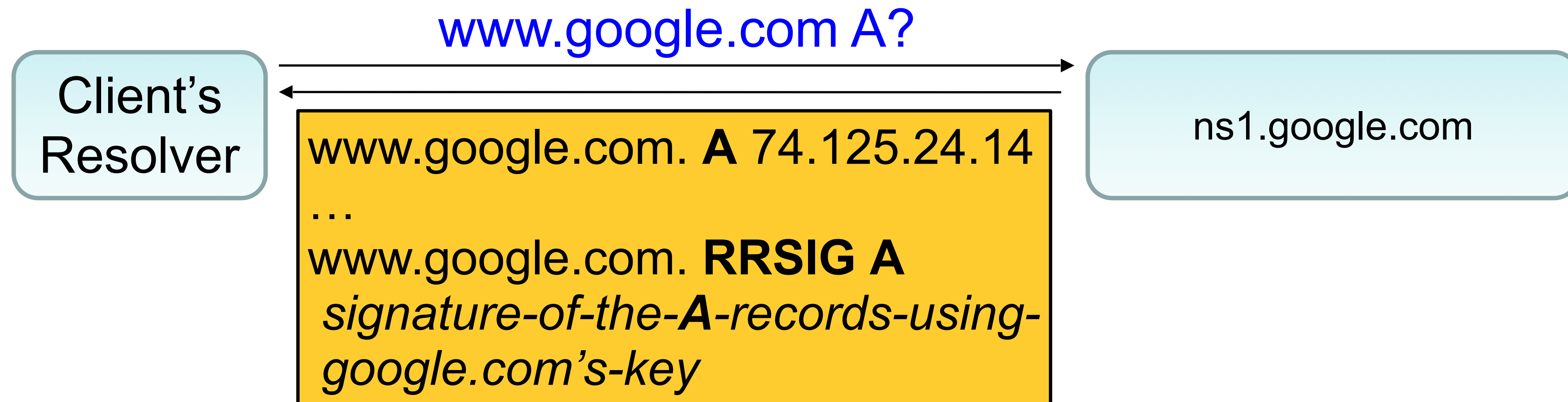
# DNSSEC (with simplifications):



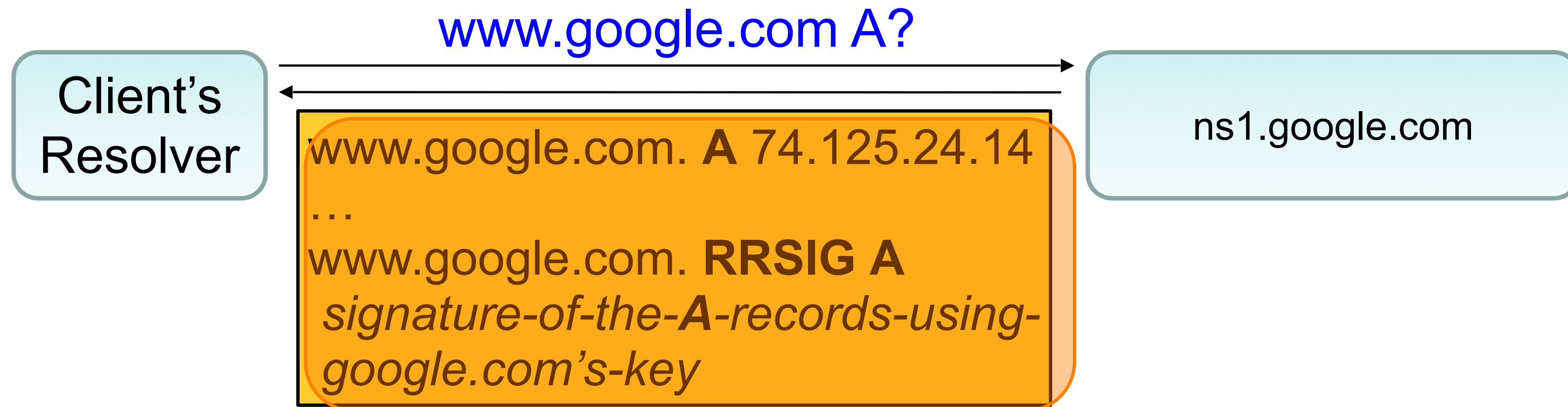
The resolver contacts one of the `google.com` name servers it's learned about.

Again, nothing guarantees this is a legitimate name server for the query!

# DNSSEC (with simplifications):



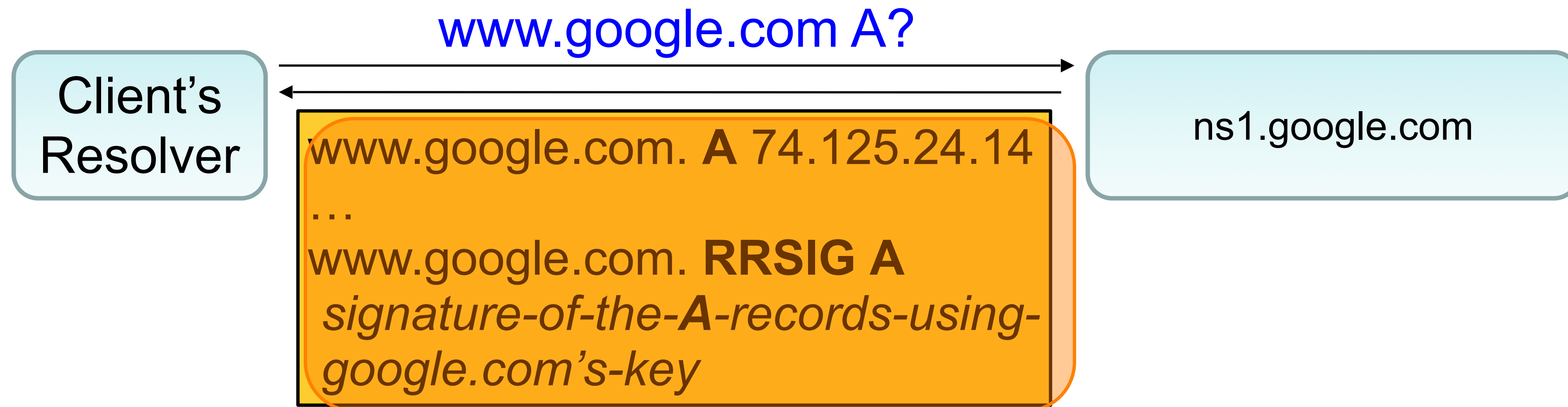
# DNSSEC (with simplifications):



Finally we've received the information we wanted (**A** records for `www.google.com`) ! ... *and* we receive a signature over those records

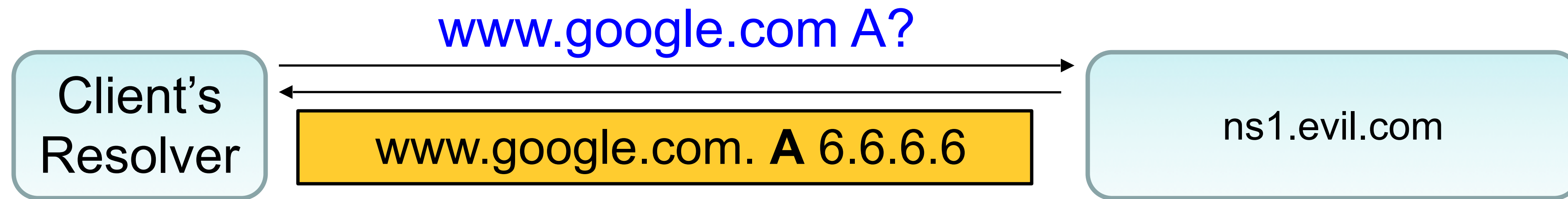


# DNSSEC (with simplifications):

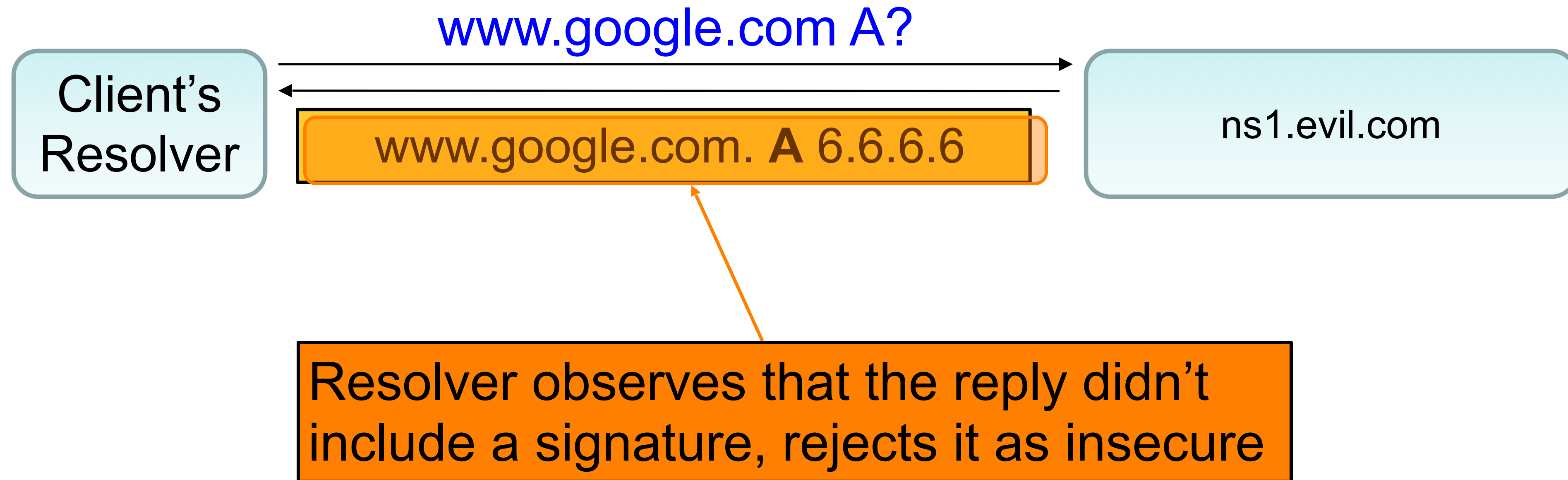


Assuming the signature validates, then because we believe (due to the signature chain) it's indeed from `google.com`'s key, we can trust that this is a correct set of **A** records ...  
Regardless of what name server returned them to us!

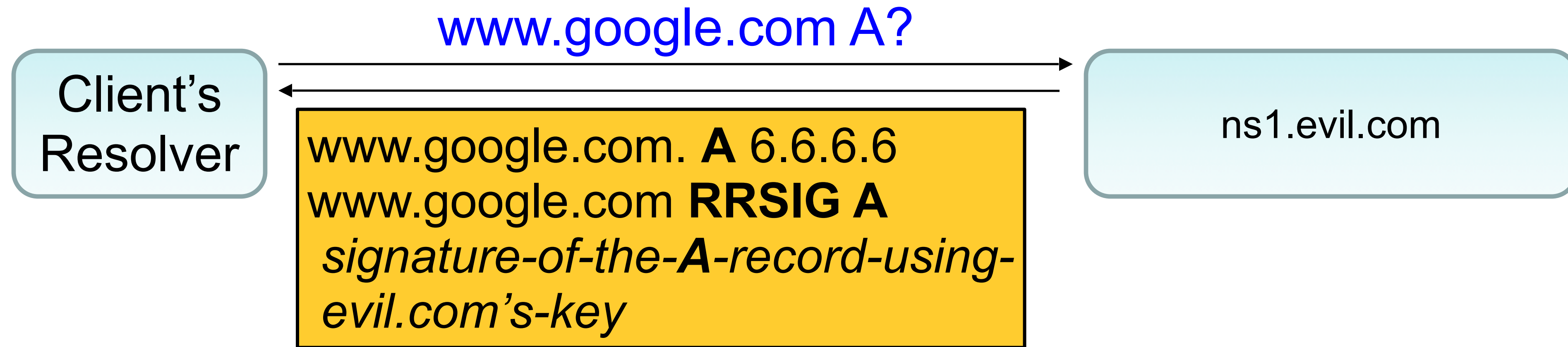
# DNSSEC – Mallory attacks!



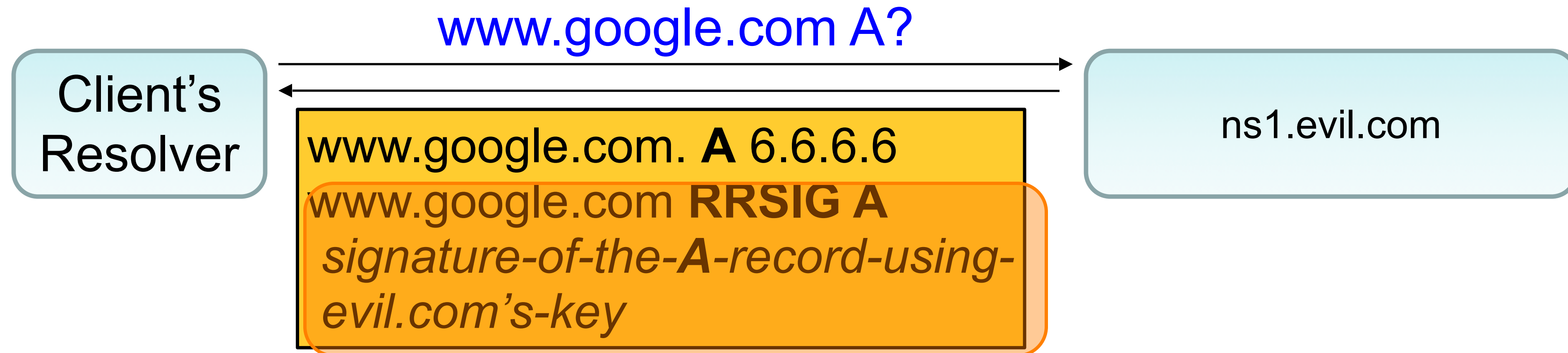
# DNSSEC – Mallory attacks!



# DNSSEC – Mallory attacks!

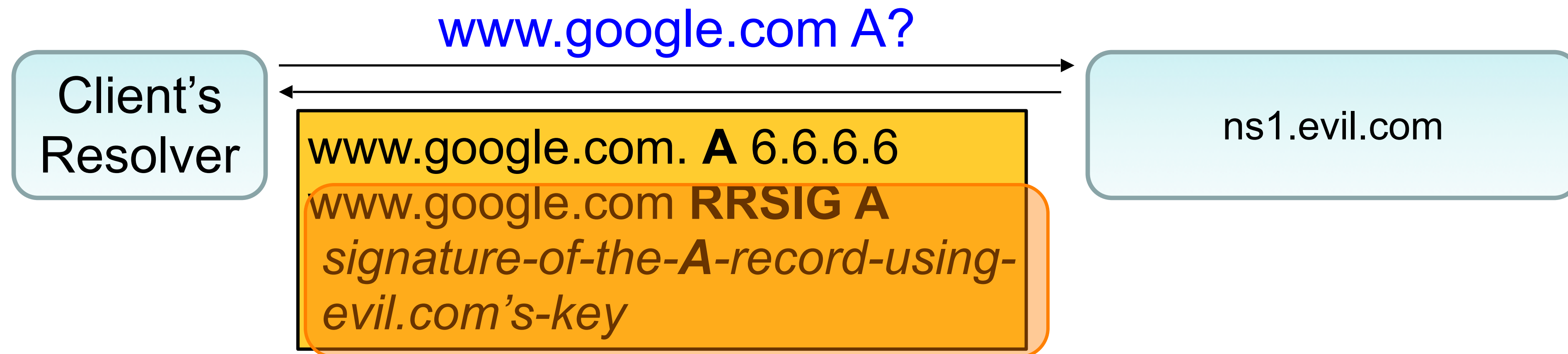


# DNSSEC – Mallory attacks!



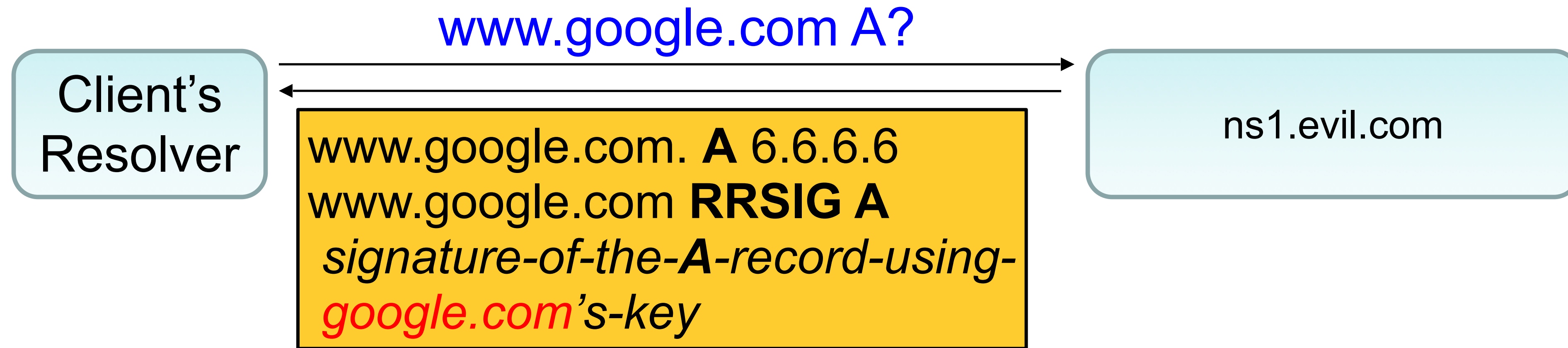
(1) If resolver didn't receive a signature from .com for evil.com's key, then it can't validate this signature & ignores reply since it's not properly signed ...

# DNSSEC – Mallory attacks!

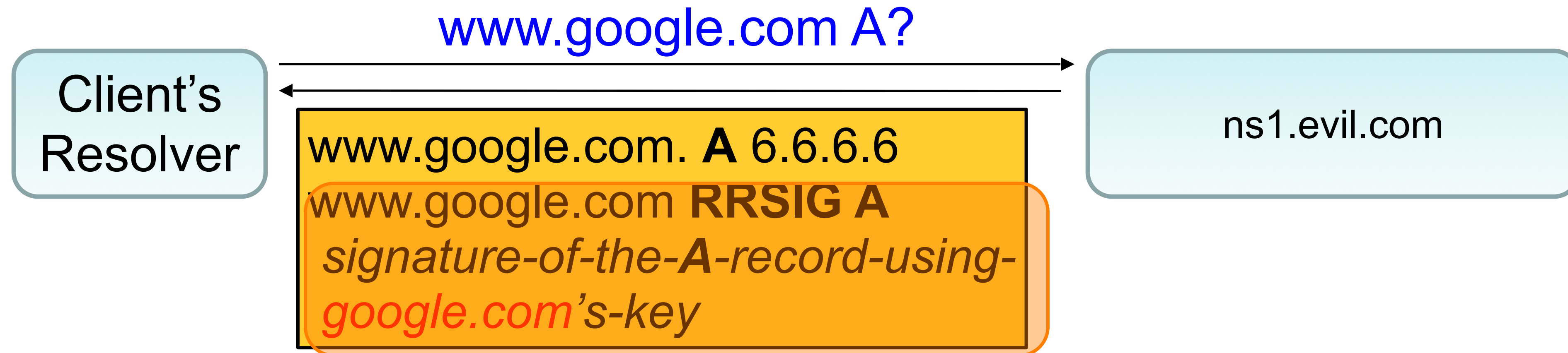


(2) If resolver *did* receive a signature from .com for evil.com's key, then it knows the key is for evil.com and not google.com ... and ignores it

# DNSSEC – Mallory attacks!



# DNSSEC – Mallory attacks!



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



# Issues With DNSSEC, cont.

- Issue #1: *Partial deployment*
  - Suppose `.com` not signing, though `google.com` is. Or, suppose `.com` and `google.com` are signing, but `cnn.com` isn't. Major practical concern. What do we do?
  - What do you do with unsigned/unvalidated results?
  - If you trust them, **weakens incentive** to upgrade (man-in-the-middle attacker can defeat security even for google.com, by sending forged but unsigned response)
  - If you don't trust them, a whole lot of things **break**

# Issues With DNSSEC, cont.

- Issue #2: Negative results (“no such name”)
  - What statement does the nameserver sign?
  - If “gab1uph.google.com” doesn’t exist, then have to do dynamic key-signing (expensive) for any bogus request
  - Instead, sign (off-line) statements about order of names
    - E.g., sign “gabby.google.com is followed by gaelic.google.com”
    - Thus, can see that gadfly.google.com can’t exist
  - But: now attacker can **enumerate** all names that exist :-)

# Issues with DNSSEC

- Issue #3: Replies are Big
  - E.g., “dig +dnssec berkeley.edu” can return 2100+ B
  - DoS **amplification**
  - Increased **latency** on low-capacity links
  - Headaches w/ older libraries that assume replies < 512B

# Adoption of DNSSEC

- Adopted, but not nearly as much as TLS
- Difficulties with deploying DNSSEC:
  - The need to design a backward-compatible standard that can scale to the size of the Internet
  - Zone enumeration attack
  - Deployment of DNSSEC implementations across a wide variety of DNS servers and resolvers (clients)
  - Disagreement among implementers over who should own the top level domain keys
  - Overcoming the perceived complexity of DNSSEC and DNSSEC deployment

# Summary of TLS & DNSSEC Technologies

- **TLS**: provides **channel security** (for communication over TCP)
  - Confidentiality, integrity, authentication
  - Client & server agree on crypto, session keys
  - Underlying security dependent on:
    - Trust in **Certificate Authorities** / decisions to sign keys
    - (as well as implementors)
- **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, and all other signing keys

# Takeaways

- Channel security vs object security
- PKI organization should follow existing line of authority