Securing Internet Communication: TLS & DNSSEC

CS 161: Computer Security

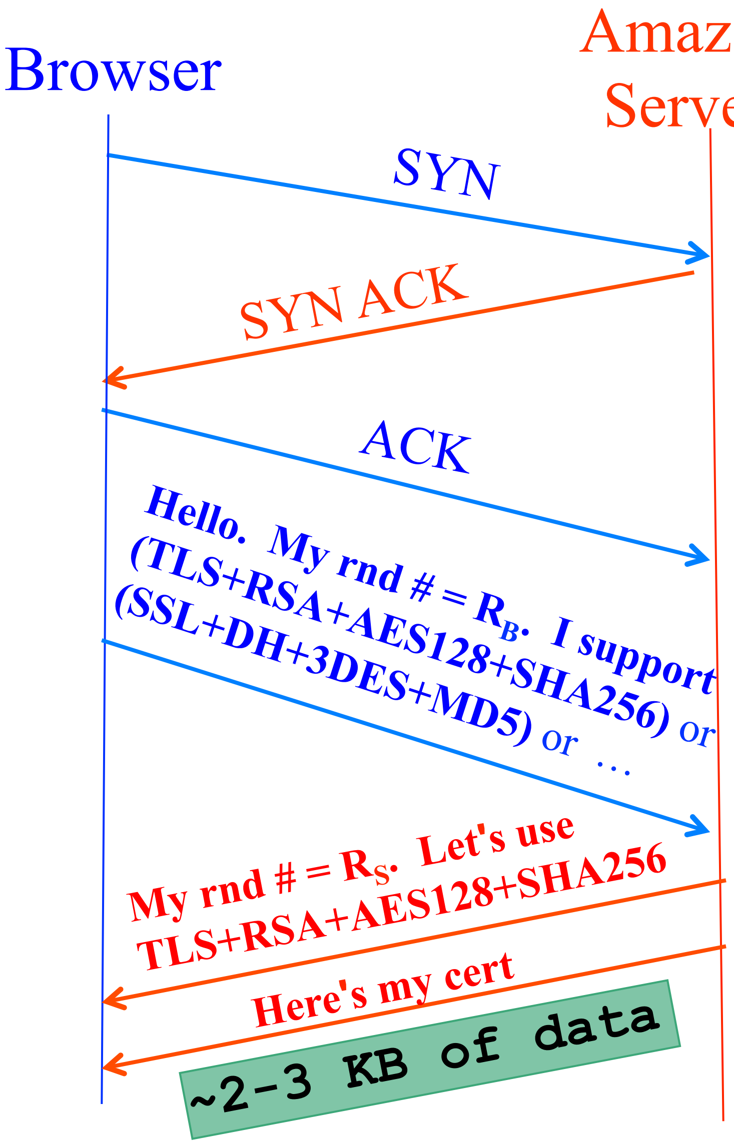
Prof. Vern Paxson

TAs: Paul Bramsen, Apoorva Dornadula,
David Fifield, Mia Gil Epner, David Hahn, Warren He,
Grant Ho, Frank Li, Nathan Malkin, Mitar Milutinovic,
Rishabh Poddar, Rebecca Portnoff, Nate Wang

https://inst.eecs.berkeley.edu/~cs161/

April 11, 2017

TLS Protocol Diagram: Q’s?


Browser

Amazon Server

SYN

SYN ACK

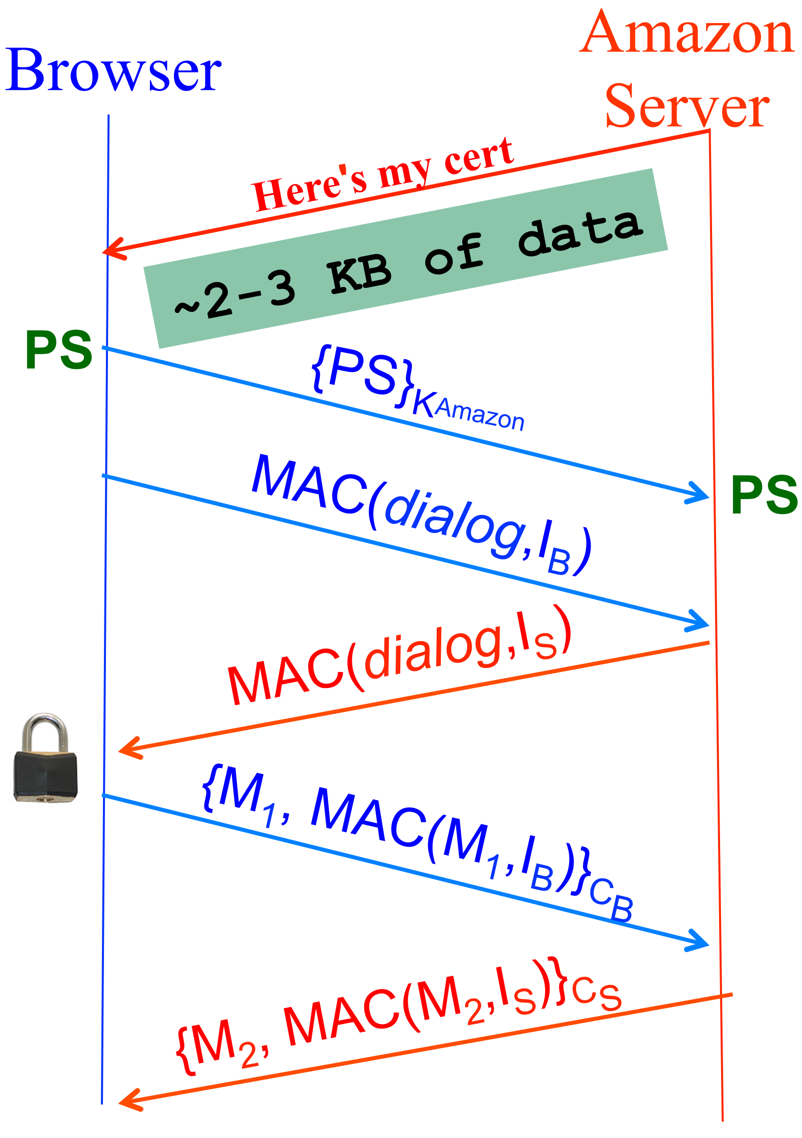
ACK

Hello. My rnd # = R_B. I support (TLS+RSA+AES128+SHA256) or (SSL+DH+3DES+MD5) or …

My rnd # = R_S. Let's use TLS+RSA+AES128+SHA256

Here's my cert

`~2-3 KB of data`


Browser

Amazon Server

Here's my cert

`~2-3 KB of data`

PS

$\{PS\}_{K_{Amazon}}$

PS

$MAC(dialog, I_B)$

$MAC(dialog, I_S)$

$\{M_1, MAC(M_1, I_B)\}_{C_B}$

$\{M_2, MAC(M_2, I_S)\}_{C_S}$

SSL / TLS Limitations

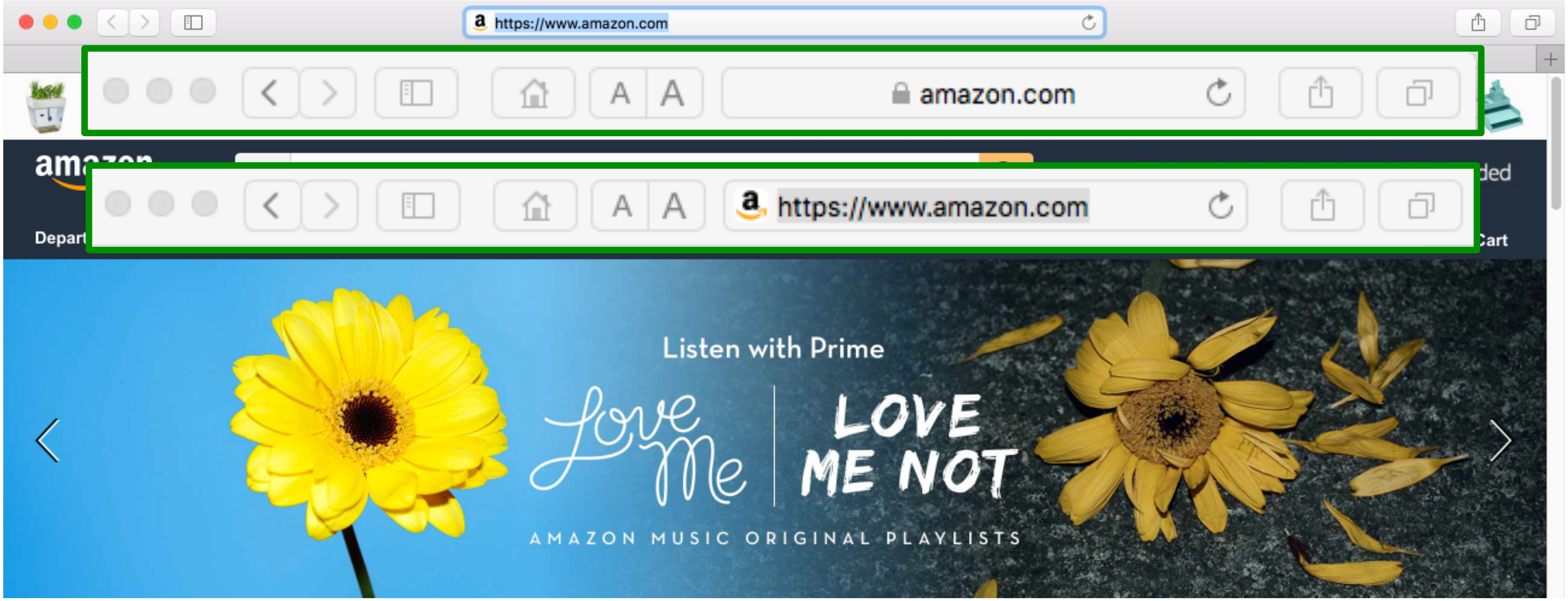
- Properly used, SSL / TLS provides powerful end-to-end protections
- So why not use it for *everything*??
- Issues:
  - Cost of public-key crypto
    - Takes non-trivial CPU processing (but today a minor issue)
    - Note: *symmetric* key crypto on modern hardware is non-issue
  - Hassle of buying/maintaining certs (fairly minor)
  - DoS amplification
    - Client can force server to undertake public key operations
    - But: requires established TCP connection, and given that, there are often other juicy targets like back-end databases
  - Integrating with other sites that don't use HTTPS
  - **Latency**: extra round trips ⇒ pages take longer to load

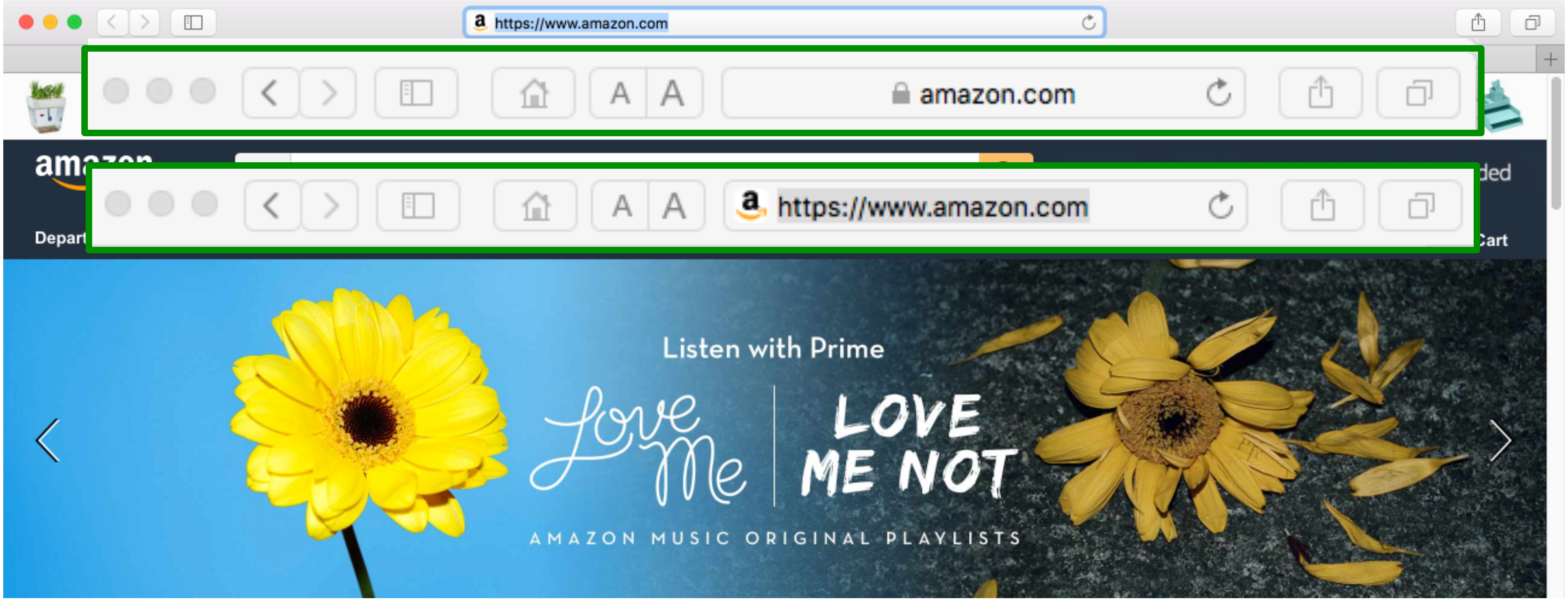
SSL / TLS Limitations, con’t

- Problems that SSL / TLS does **not** take care of ?
- TCP-level denial of service (or any other DoS)
  - SYN flooding
  - RST injection
    - (but does protect against data injection!)
- SQL injection / XSS / server-side coding/logic flaws
- Browser coding/logic flaws
- User flaws
  - Weak passwords
  - Phishing
- Vulnerabilities introduced by HTTP compatibility …

www.amazon.com

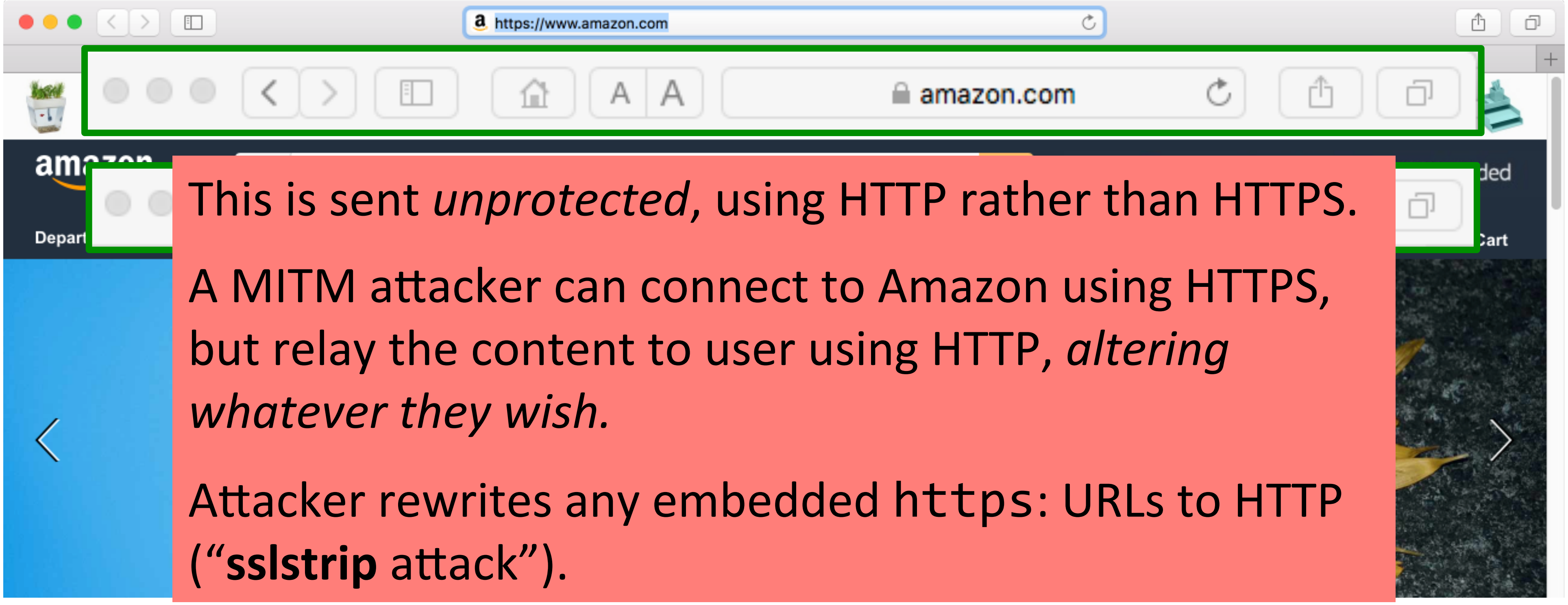
```
GET / HTTP/1.1
Host: www.amazon.com
Cookie: ...

HTTP/1.1 301 Moved Permanently
Location: https://www.amazon.com/
```


https://www.amazon.com
amazon.com
https://www.amazon.com
A A
Listen with Prime
Love Me
LOVE ME NOT
AMAZON MUSIC ORIGINAL PLAYLISTS



~~GET / HTTP/1.1~~
~~Host: www.amazon.com~~
~~Cookie: ...~~

~~HTTP/1.1 301 Moved Permanently~~
~~Location: https://www.amazon.com/~~



This is sent *unprotected*, using HTTP rather than HTTPS.

A MITM attacker can connect to Amazon using HTTPS, but relay the content to user using HTTP, *altering whatever they wish.*

Attacker rewrites any embedded `https:` URLs to HTTP (“**sslstrip** attack”).

~~GET / HTTP/1.1~~
~~Host: www.amazon.com~~
~~Cookie: ...~~

~~HTTP/1.1 301 Moved Permanently~~
~~Location: https://www.amazon.com/~~

HTTP *Strict Transport Security*

- To defend against **sslstrip** attacks, a web server can return (during HTTPS conn.) directive such as:

  `Strict-Transport-Security: max-age=31536000 includeSubDomains`

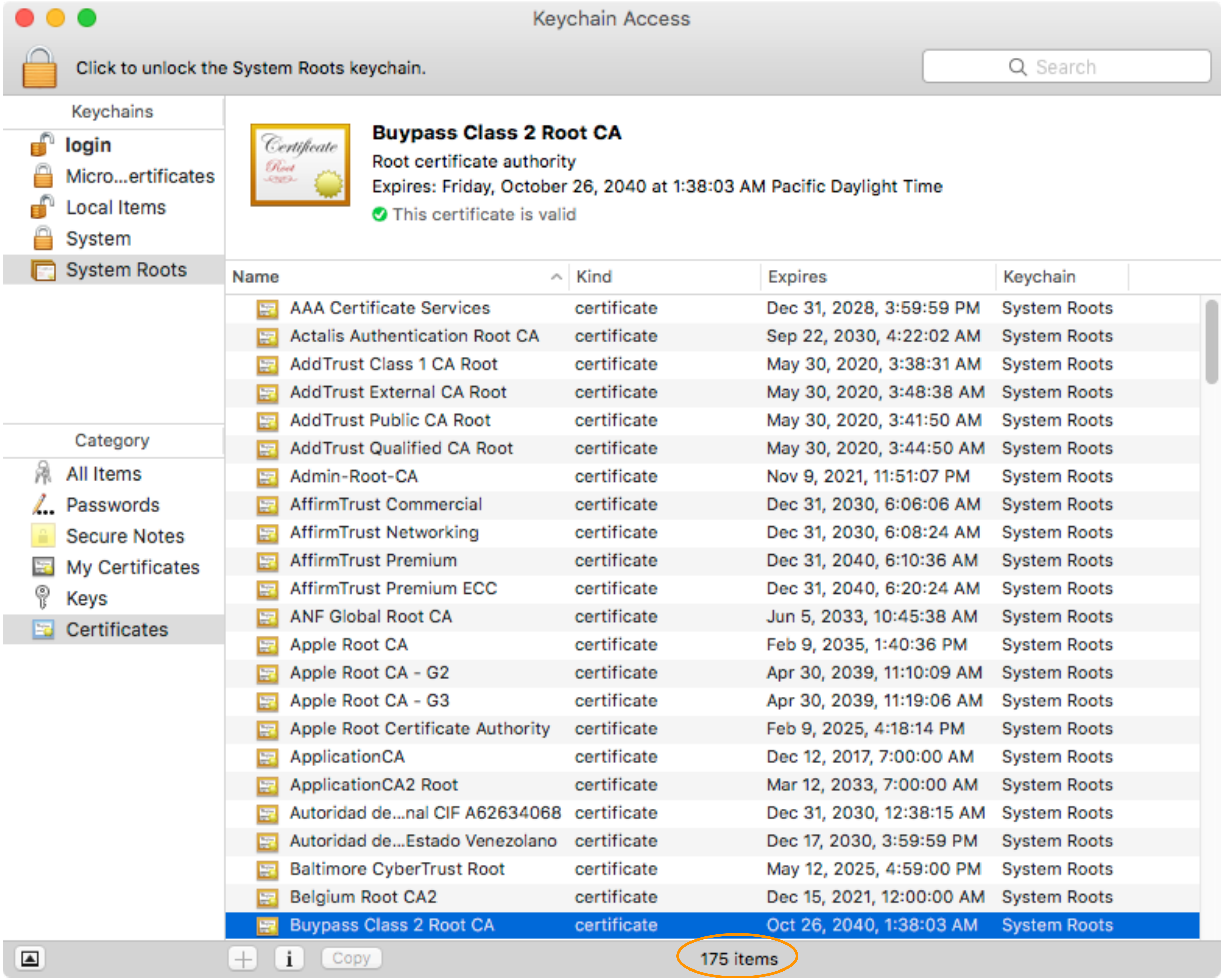
- Directs browser to:
  - Only connect to that site using HTTPS (expires in 1yr)
  - Promote any HTTP links in pages to HTTPS
  - Don't allow connections w/ cert errors to proceed
- Similar to TOFU, requires safe initial connection
  - Otherwise, MITM attacker could strip out the header
- Many browsers today use a predefined list of HSTS sites – see `https://hstspreload.org/`

SSL / TLS Limitations, con't

- Problems that SSL / TLS does **not** take care of ?
- TCP-level denial of service
  - SYN flooding
  - RST injection
    - (but does protect against data injection!)
- SQL injection / XSS / server-side coding/logic flaws
- Browser coding/logic flaws
- User flaws
  - Weak passwords
  - Phishing
- Vulnerabilities introduced by HTTP compatibility …
- Issues of trust …

TLS/SSL Trust Issues

- "*Commercial certificate authorities protect you from anyone from whom they are unwilling to take money*"
  - Matt Blaze, circa 2001
- So how many CAs do we have to worry about, anyway?



Click to unlock the System Roots keychain.

Buypass Class 2 Root CA
Root certificate authority
Expires: Friday, October 26, 2040 at 1:38:03 AM Pacific Daylight Time
✓ This certificate is valid

Name	Kind	Expires	Keychain
AAA Certificate Services	certificate	Dec 31, 2028, 3:59:59 PM	System Roots
Actalis Authentication Root CA	certificate	Sep 22, 2030, 4:22:02 AM	System Roots
AddTrust Class 1 CA Root	certificate	May 30, 2020, 3:38:31 AM	System Roots
AddTrust External CA Root	certificate	May 30, 2020, 3:48:38 AM	System Roots
AddTrust Public CA Root	certificate	May 30, 2020, 3:41:50 AM	System Roots
AddTrust Qualified CA Root	certificate	May 30, 2020, 3:44:50 AM	System Roots
Admin-Root-CA	certificate	Nov 9, 2021, 11:51:07 PM	System Roots
AffirmTrust Commercial	certificate	Dec 31, 2030, 6:06:06 AM	System Roots
AffirmTrust Networking	certificate	Dec 31, 2030, 6:08:24 AM	System Roots
AffirmTrust Premium	certificate	Dec 31, 2040, 6:10:36 AM	System Roots
AffirmTrust Premium ECC	certificate	Dec 31, 2040, 6:20:24 AM	System Roots
ANF Global Root CA	certificate	Jun 5, 2033, 10:45:38 AM	System Roots
Apple Root CA	certificate	Feb 9, 2035, 1:40:36 PM	System Roots
Apple Root CA - G2	certificate	Apr 30, 2039, 11:10:09 AM	System Roots
Apple Root CA - G3	certificate	Apr 30, 2039, 11:19:06 AM	System Roots
Apple Root Certificate Authority	certificate	Feb 9, 2025, 4:18:14 PM	System Roots
ApplicationCA	certificate	Dec 12, 2017, 7:00:00 AM	System Roots
ApplicationCA2 Root	certificate	Mar 12, 2033, 7:00:00 AM	System Roots
Autoridad de...nal CIF A62634068	certificate	Dec 31, 2030, 12:38:15 AM	System Roots
Autoridad de...Estado Venezolano	certificate	Dec 17, 2030, 3:59:59 PM	System Roots
Baltimore CyberTrust Root	certificate	May 12, 2025, 4:59:00 PM	System Roots
Belgium Root CA2	certificate	Dec 15, 2021, 12:00:00 AM	System Roots
Buypass Class 2 Root CA	certificate	Oct 26, 2040, 1:38:03 AM	System Roots

175 items

TLS/SSL Trust Issues

- "*Commercial certificate authorities protect you from anyone from whom they are unwilling to take money*"
  - Matt Blaze, circa 2001
- So how many CAs do we have to worry about, anyway?
- Of course, it's not just their greed that matters …

News

Solo Iranian hacker takes credit for Comodo certificate attack

Security researchers split on whether 'ComodoHacker' is the real deal

By Gregg Keizer
March 27, 2011 08:39 PM ET

Computerworld - A solo Iranian hacker on Saturday claimed responsibility for stealing multiple SSL certificates belonging to some of the Web's biggest sites, including Google, Microsoft, Skype and Yahoo.

Early reaction from security experts was mixed, with some believing the hacker's claim, while others were dubious.

Last week, conjecture had focused on a state-sponsored attack, perhaps funded or conducted by the Iranian government, that hacked a certificate reseller affiliated with U.S.-based Comodo.

On March 23, Comodo acknowledged the attack, saying that eight days earlier, hackers had obtained nine bogus certificates for the log-on sites of Microsoft's Hotmail, Google's Gmail, the Internet phone and chat service Skype and Yahoo Mail. A certificate for Mozilla's Firefox add-on site was also acquired.

News

Solo Iranian hacker takes credit for Comodo certificate attack

Security researchers split on whether 'ComodoHacker' is the real deal

By Gregg Keizer

March 27, 2011 08:39 PM ET

Where did you learn about cryptography and hacking. Are there books in Persian? English books? Or are you self-taught, learning from the Internet?

> d) I'm self taught, books in Persian and English, but mostly papers in internet, short papers from experts like Bruce Schneier, RSA people (Ron, Adi and Leonard) and specially . I learned programming in Qbasic when I was 9, I started learning cryptography when I was 13

funded or conducted by the Iranian government, that hacked a certificate reseller affiliated with U.S.-based Comodo.

On March 23, Comodo acknowledged the attack, saying that eight days earlier, hackers had obtained nine bogus certificates for the log-on sites of Microsoft's Hotmail, Google's Gmail, the Internet phone and chat service Skype and Yahoo Mail. A certificate for Mozilla's Firefox add-on site was also acquired.

CNET › News › InSecurity Complex › Fraudulent Google certificate points to Internet attack

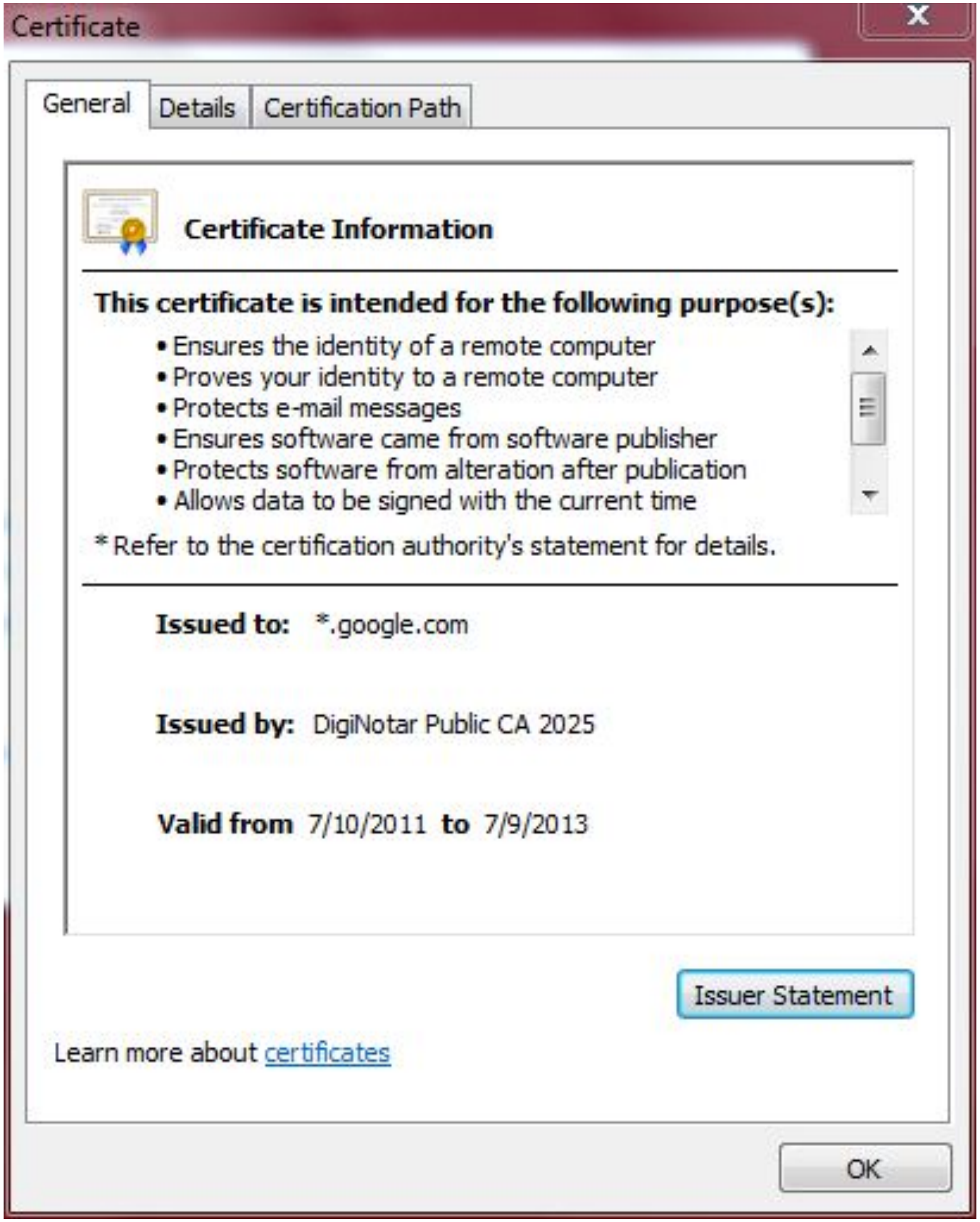
Fraudulent Google certificate points to Internet attack

Is Iran behind a fraudulent Google.com digital certificate? The situation is similar to one that happened in March in which spoofed certificates were traced back to Iran.

by Elinor Mills | August 29, 2011 1:22 PM PDT

A Dutch company appears to have issued a digital certificate for Google.com to someone other than Google, who may be using it to try to re-direct traffic of users based in Iran.

Yesterday, someone reported on a Google support site that when attempting to log in to Gmail the browser issued a warning for the digital certificate used as proof that the site is legitimate, according to this thread on a Google support forum site.



Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of a remote computer
- Proves your identity to a remote computer
- Protects e-mail messages
- Ensures software came from software publisher
- Protects software from alteration after publication
- Allows data to be signed with the current time

* Refer to the certification authority's statement for details.

Issued to: *.google.com

Issued by: DigiNotar Public CA 2025

Valid from 7/10/2011 **to** 7/9/2013

Learn more about certificates

This appears to be a fully valid cert using normal browser validation rules.

Only detected by Chrome due to its recent introduction of cert “pinning” - requiring that certs for certain domains **must** be signed by specific CAs rather than any generally trusted CA.

October 31, 2012, 10:49AM

Final Report on DigiNotar Hack Shows Total Compromise of CA Servers

The attacker who penetrated the Dutch CA DigiNotar last year had complete control of all eight of the company's certificate-issuing servers during the operation and he may also have issued some rogue certificates that have not yet been identified. The final report from a

Evidence Suggests DigiNotar, Who Issued Fraudulent Google Certificate, Was Hacked *Years* Ago

from the *diginot* dept

The big news in the security world, obviously, is the fact that a **fraudulent Google certificate made its way out into the wild**, apparently targeting internet users in Iran. The Dutch company DigiNotar has put out a statement saying that **it discovered a breach** back on July 19th during a security audit, and that fraudulent certificates were generated for "several dozen" websites. The only one known to have gotten out into the wild is the Google one.

TLS/SSL Trust Issues

- "*Commercial certificate authorities protect you from anyone from whom they are unwilling to take money*"
  - Matt Blaze, circa 2001
- So how many CAs do we have to worry about, anyway?
- Of course, it's not just their greed that matters …
- … and it's not just their diligence & security that matters …
  - "*A decade ago, I observed that commercial certificate authorities protect you from anyone from whom they are unwilling to take money. That turns out to be wrong; they don't even do that much.*" - Matt Blaze, circa 2010

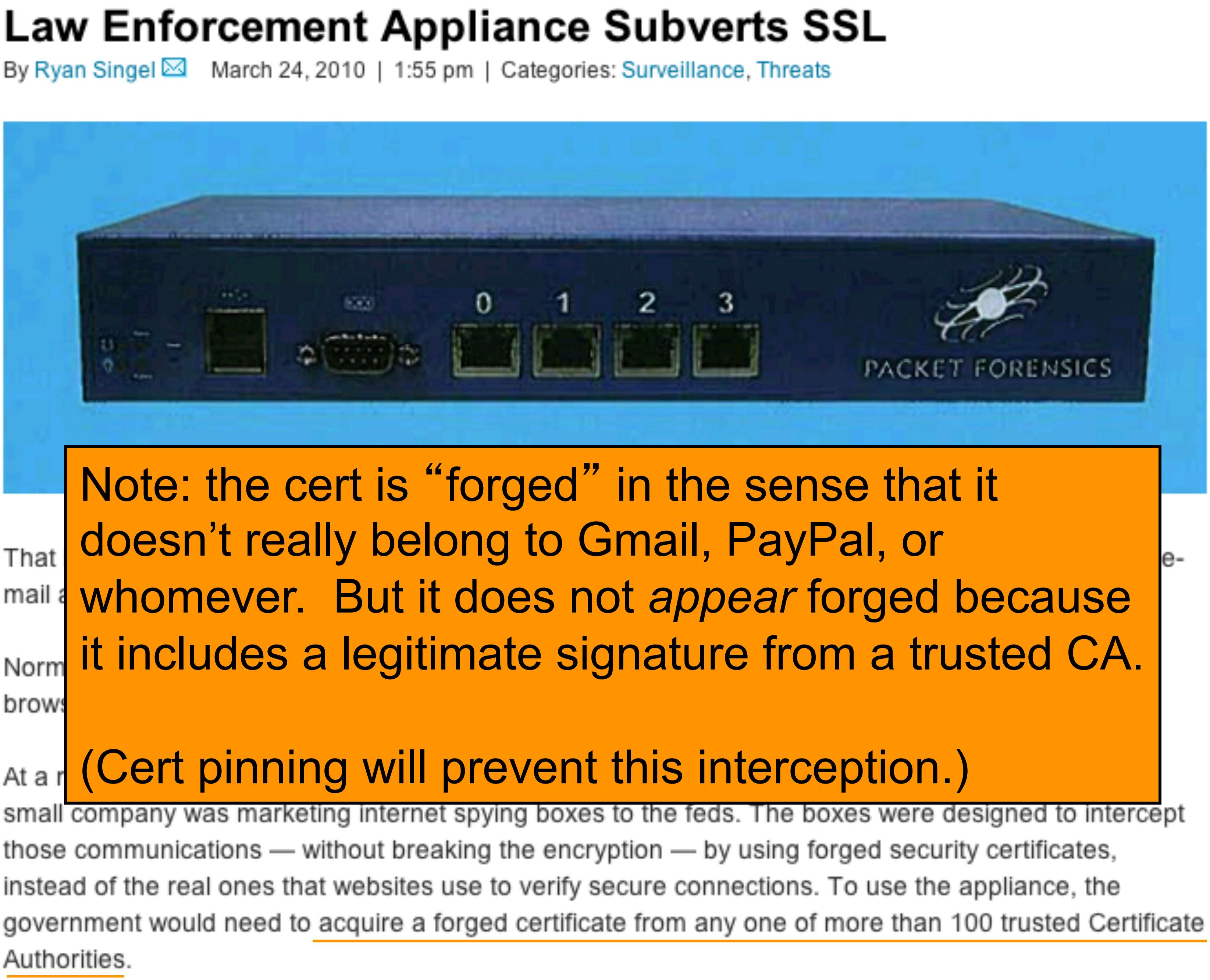
Law Enforcement Appliance Subverts SSL

By Ryan Singel March 24, 2010 | 1:55 pm | Categories: Surveillance, Threats

That little lock on your browser window indicating you are communicating securely with your bank or e-mail account may not always mean what you think its means.

Normally when a user visits a secure website, such as Bank of America, Gmail, PayPal or eBay, the browser examines the website's certificate to verify its authenticity.

At a recent wiretapping convention, however, security researcher Chris Soghoian discovered that a small company was marketing internet spying boxes to the feds. The boxes were designed to intercept those communications — without breaking the encryption — by using forged security certificates, instead of the real ones that websites use to verify secure connections. To use the appliance, the government would need to acquire a forged certificate from any one of more than 100 trusted Certificate Authorities.



Law Enforcement Appliance Subverts SSL

By Ryan Singel ✉ March 24, 2010 | 1:55 pm | Categories: Surveillance, Threats

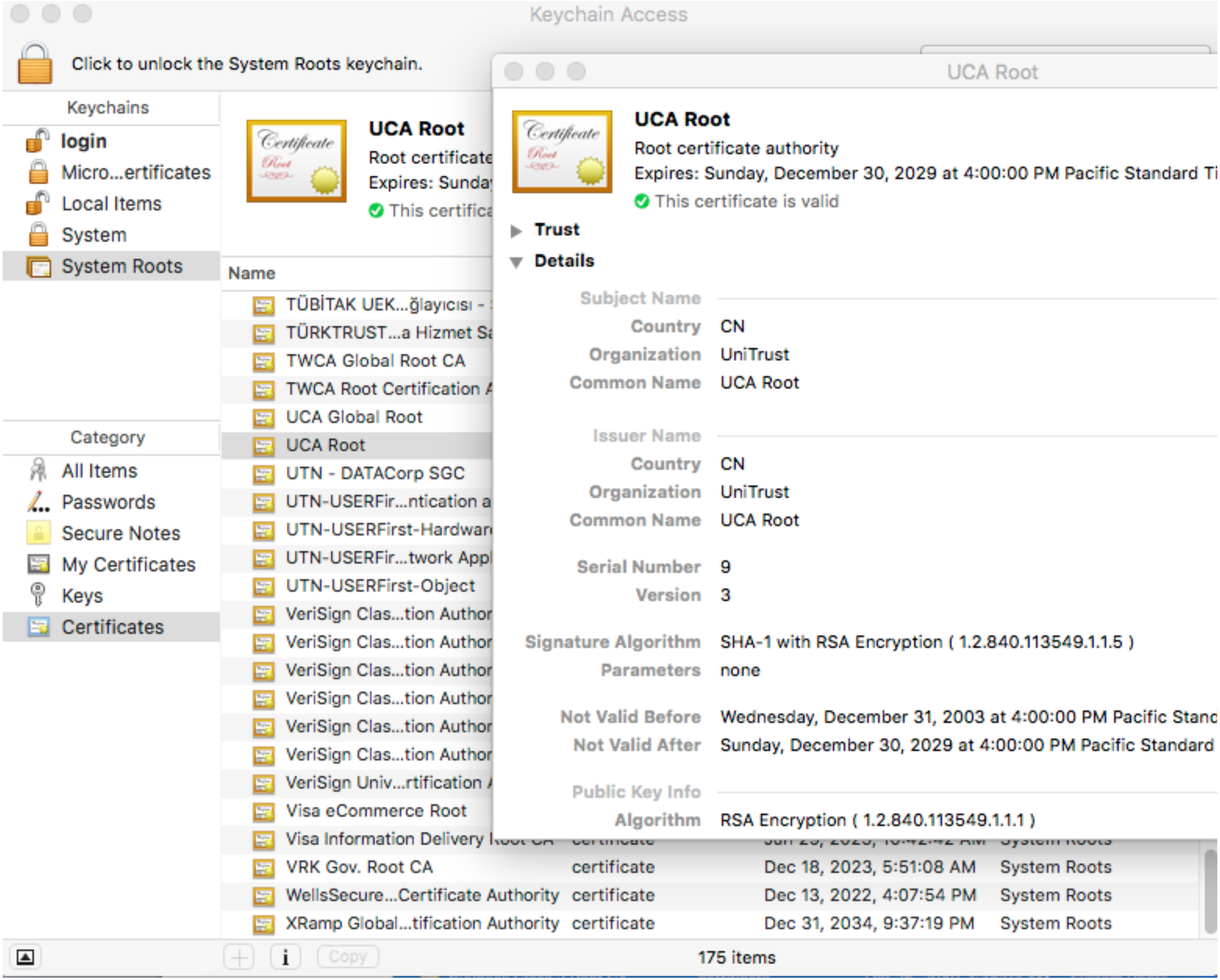
That e-
mail

Norm
brows

At a
small company was marketing internet spying boxes to the feds. The boxes were designed to intercept those communications — without breaking the encryption — by using forged security certificates, instead of the real ones that websites use to verify secure connections. To use the appliance, the government would need to acquire a forged certificate from any one of more than 100 trusted Certificate Authorities.

Note: the cert is “forged” in the sense that it doesn’t really belong to Gmail, PayPal, or whomever. But it does not *appear* forged because it includes a legitimate signature from a trusted CA.

(Cert pinning will prevent this interception.)



VRK Gov. Root CA certificate Dec 18, 2023, 5:51:08 AM System Roots

WellsSecure...Certificate Authority certificate Dec 13, 2022, 4:07:54 PM System Roots

XRamp Global...tification Authority certificate Dec 31, 2034, 9:37:19 PM System Roots

UCA Root

Root certificate authority

Expires: Sunday, December 30, 2029 at 4:00:00 PM Pacific Standard Ti

✓ This certificate is valid

Subject Name

Country CN

Organization UniTrust

Common Name UCA Root

Issuer Name

Country CN

Organization UniTrust

Common Name UCA Root

Serial Number 9

Version 3

Signature Algorithm SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)

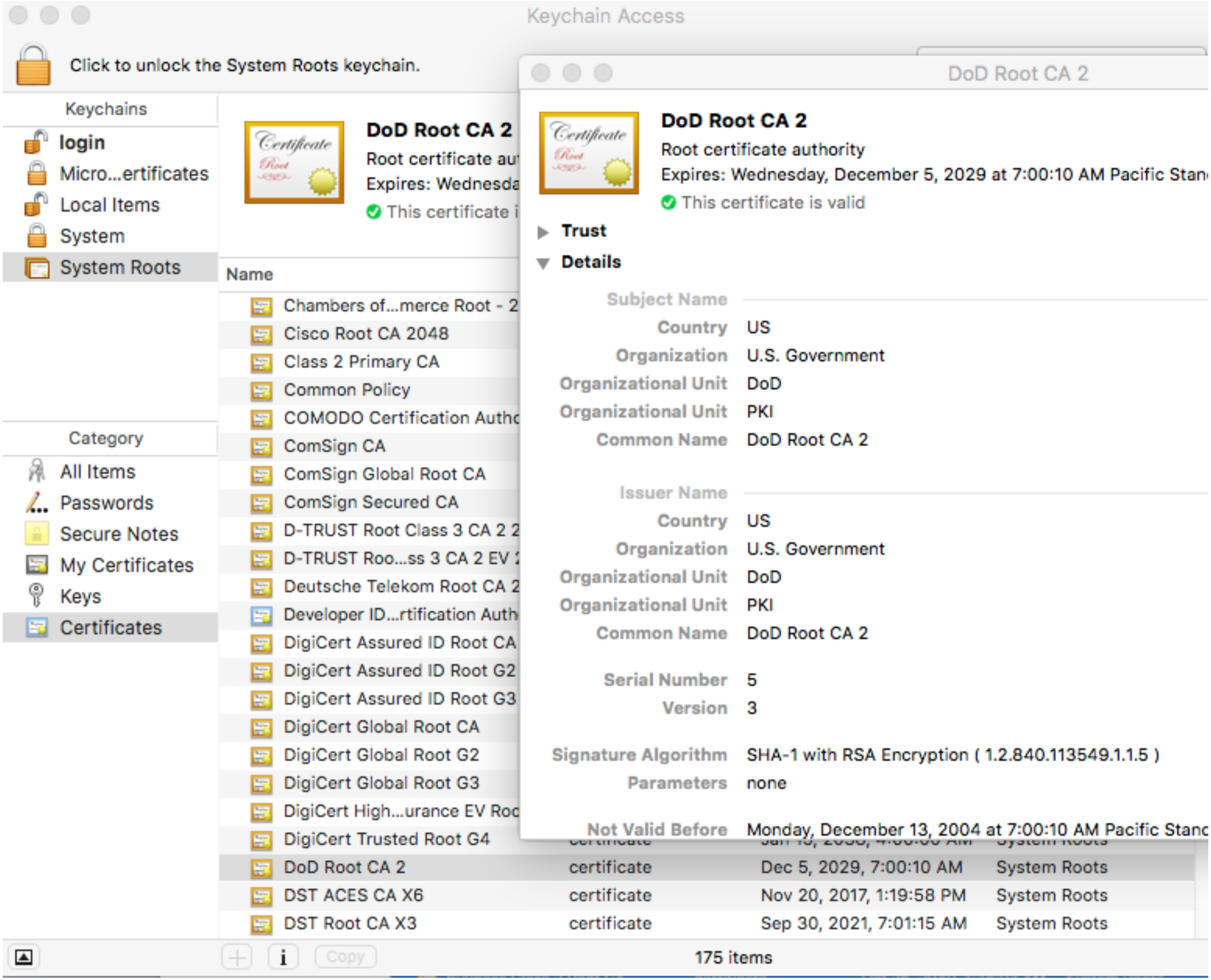
Parameters none

Not Valid Before Wednesday, December 31, 2003 at 4:00:00 PM Pacific Stand

Not Valid After Sunday, December 30, 2029 at 4:00:00 PM Pacific Standard

Public Key Info

Algorithm RSA Encryption (1.2.840.113549.1.1.1)



Chambers of...merce Root - 2
Cisco Root CA 2048
Class 2 Primary CA
Common Policy
COMODO Certification Autho
ComSign CA
ComSign Global Root CA
ComSign Secured CA
D-TRUST Root Class 3 CA 2 2
D-TRUST Roo...ss 3 CA 2 EV 2
Deutsche Telekom Root CA 2
Developer ID...rtification Auth
DigiCert Assured ID Root CA
DigiCert Assured ID Root G2
DigiCert Assured ID Root G3
DigiCert Global Root CA
DigiCert Global Root G2
DigiCert Global Root G3
DigiCert High...urance EV Roo
DigiCert Trusted Root G4

DoD Root CA 2	certificate	Dec 5, 2029, 7:00:10 AM	System Roots
DST ACES CA X6	certificate	Nov 20, 2017, 1:19:58 PM	System Roots
DST Root CA X3	certificate	Sep 30, 2021, 7:01:15 AM	System Roots

DoD Root CA 2
Root certificate authority
Expires: Wednesday, December 5, 2029 at 7:00:10 AM Pacific Stan
✓ This certificate is valid

Subject Name	
Country	US
Organization	U.S. Government
Organizational Unit	DoD
Organizational Unit	PKI
Common Name	DoD Root CA 2
Issuer Name	
Country	US
Organization	U.S. Government
Organizational Unit	DoD
Organizational Unit	PKI
Common Name	DoD Root CA 2
Serial Number	5
Version	3
Signature Algorithm	SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)
Parameters	none
Not Valid Before	Monday, December 13, 2004 at 7:00:10 AM Pacific Stanc

TLS/SSL Trust Issues

- *"Commercial certificate authorities protect you from anyone from whom they are unwilling to take money"*
  - Matt Blaze, circa 2001
- So how many CAs do we have to worry about, anyway?
- Of course, it's not just their greed that matters …
- … and it's not just their diligence & security that matters …
  - *"A decade ago, I observed that commercial certificate authorities protect you from anyone from whom they are unwilling to take money. That turns out to be wrong; they don't even do that much."* - Matt Blaze, circa 2010
- You also have to trust the developers of libraries …
  - Both for clients when validating certs …

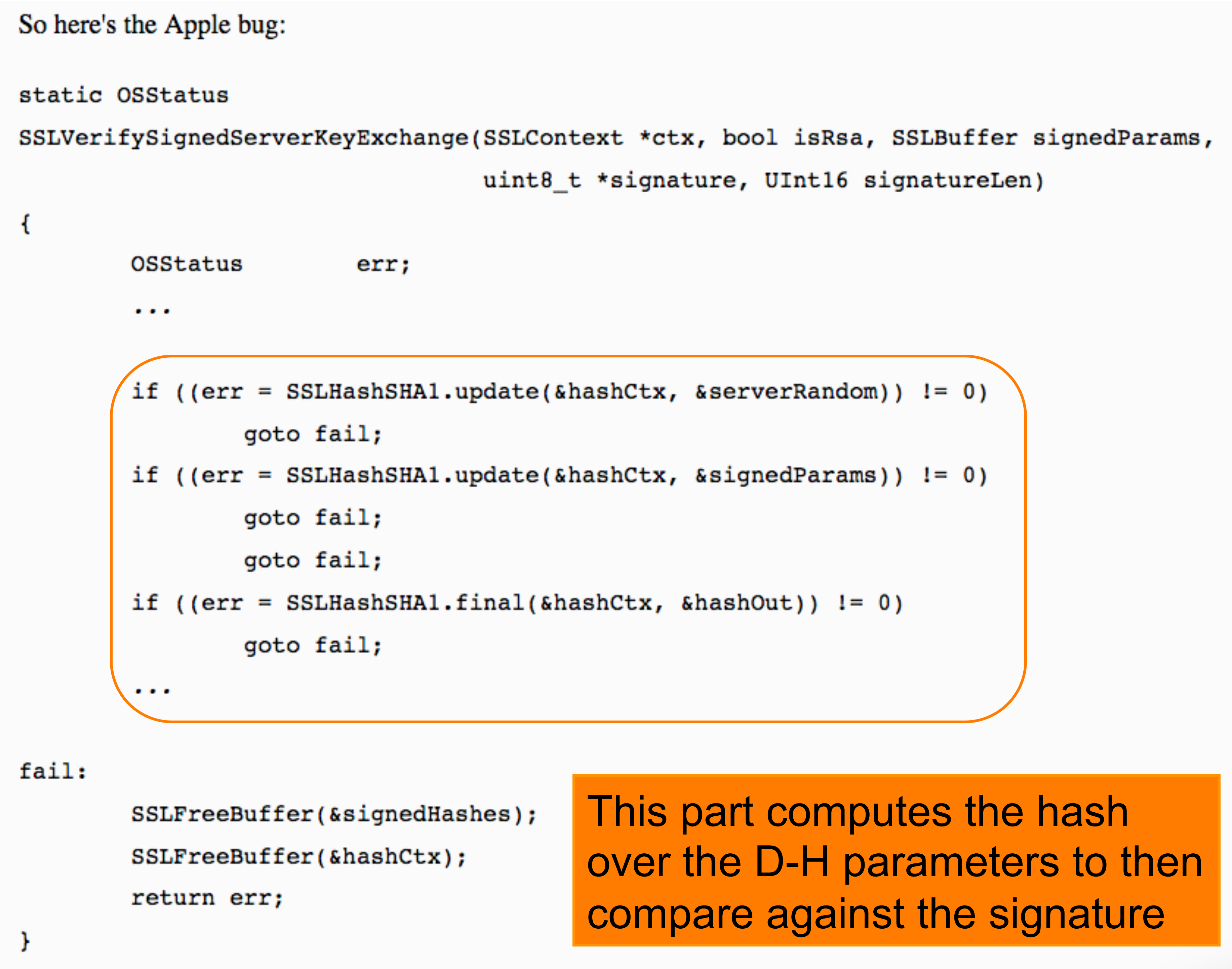
So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHas
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

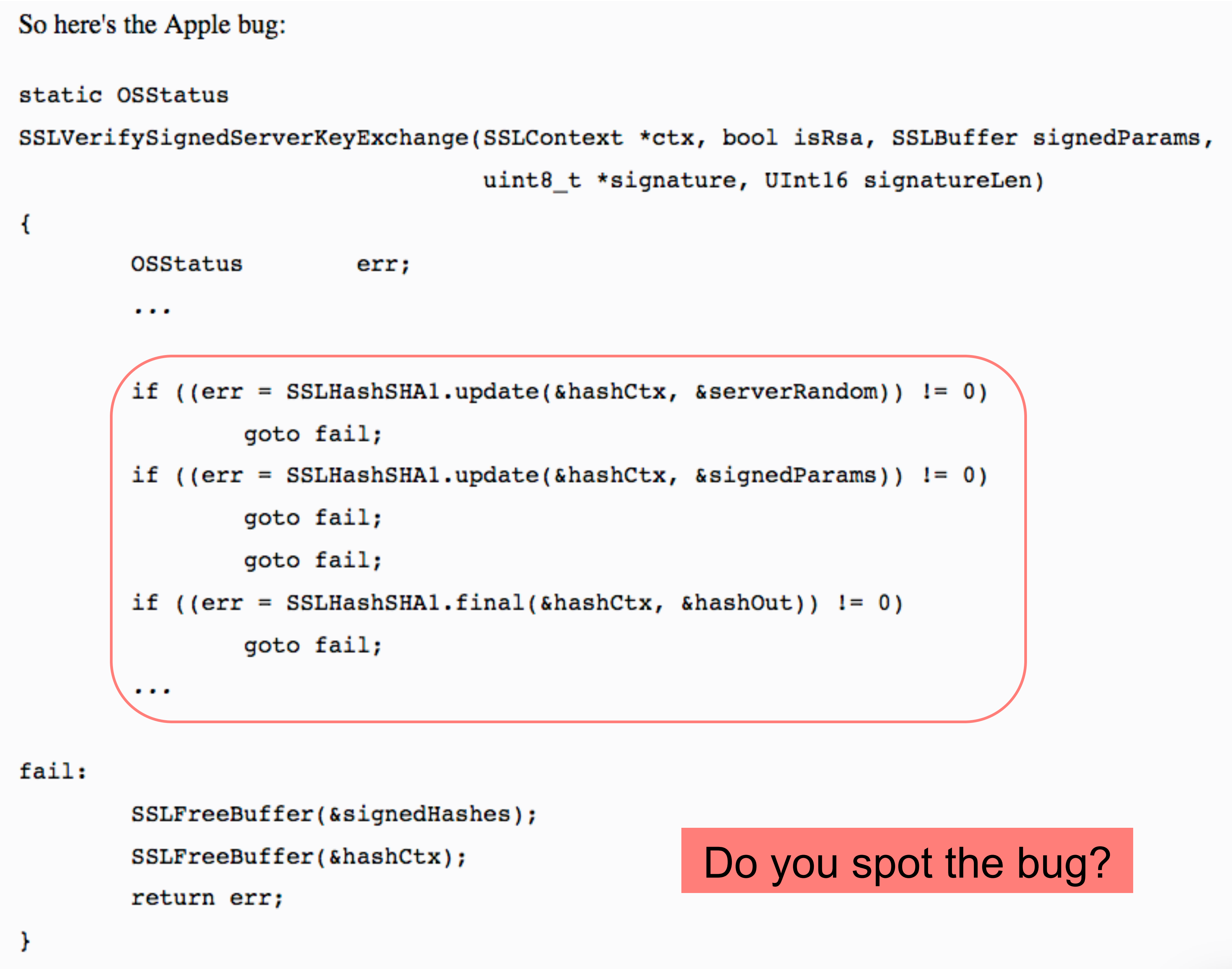
This is the code that verifies that the Diffie-Hellman parameters sent by the server have a valid signature per the public key in the server’s cert



So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

This part computes the hash over the D-H parameters to then compare against the signature



So here's the Apple bug:

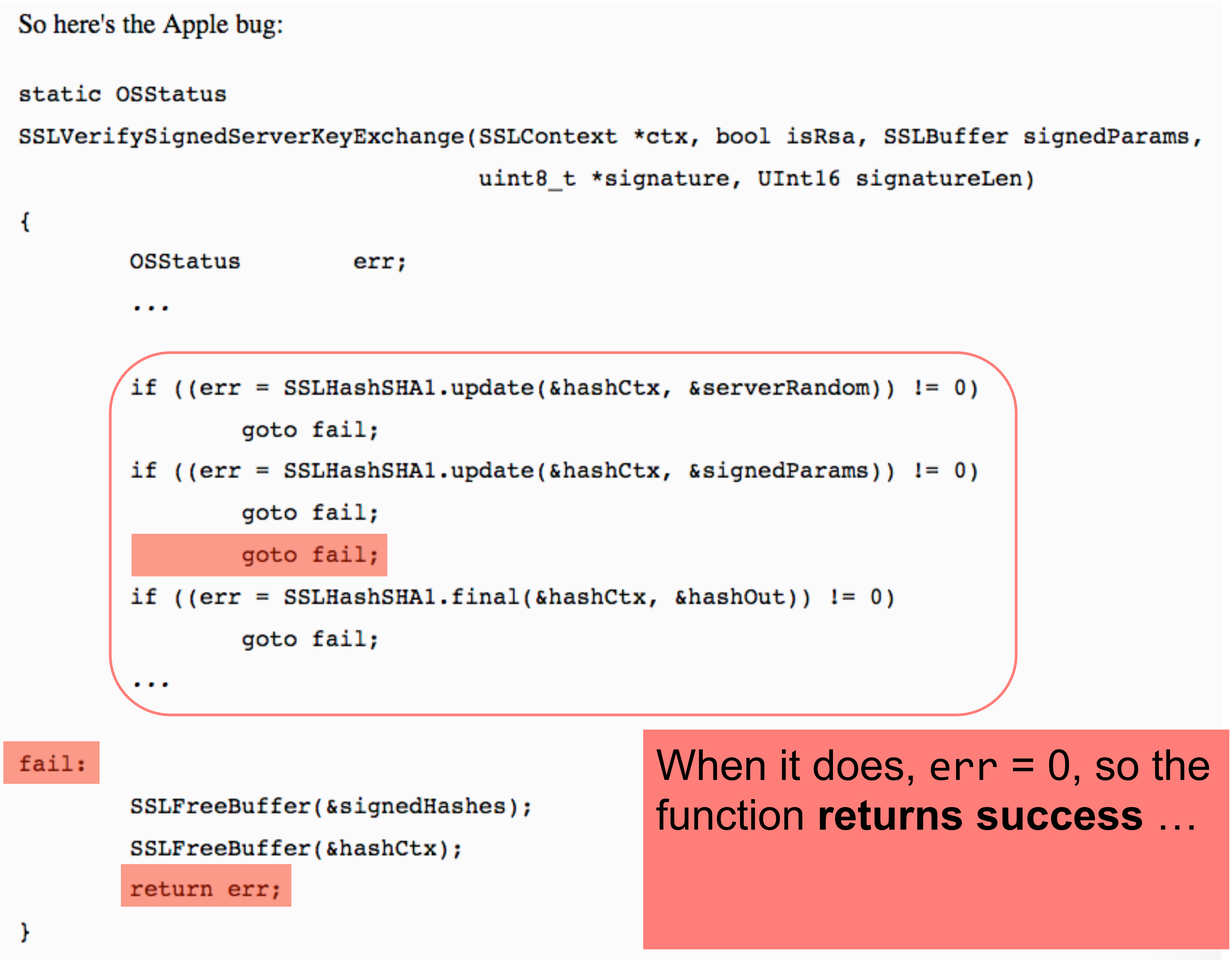
```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

Do you spot the bug?

So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

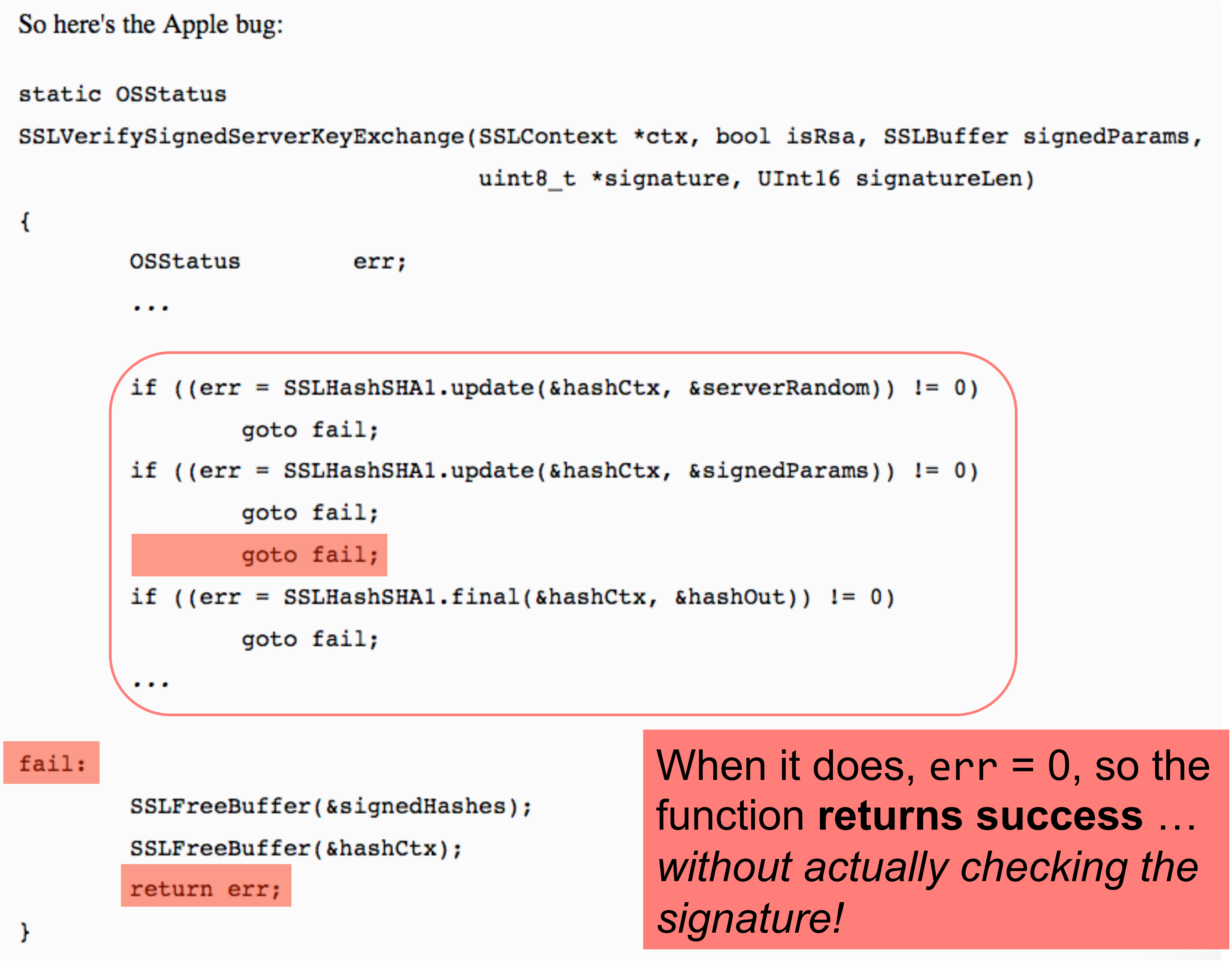
This code *always executes*!



So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

When it does, `err` = 0, so the function **returns success** …



So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

When it does, `err` = 0, so the function **returns success** … *without actually checking the signature!*

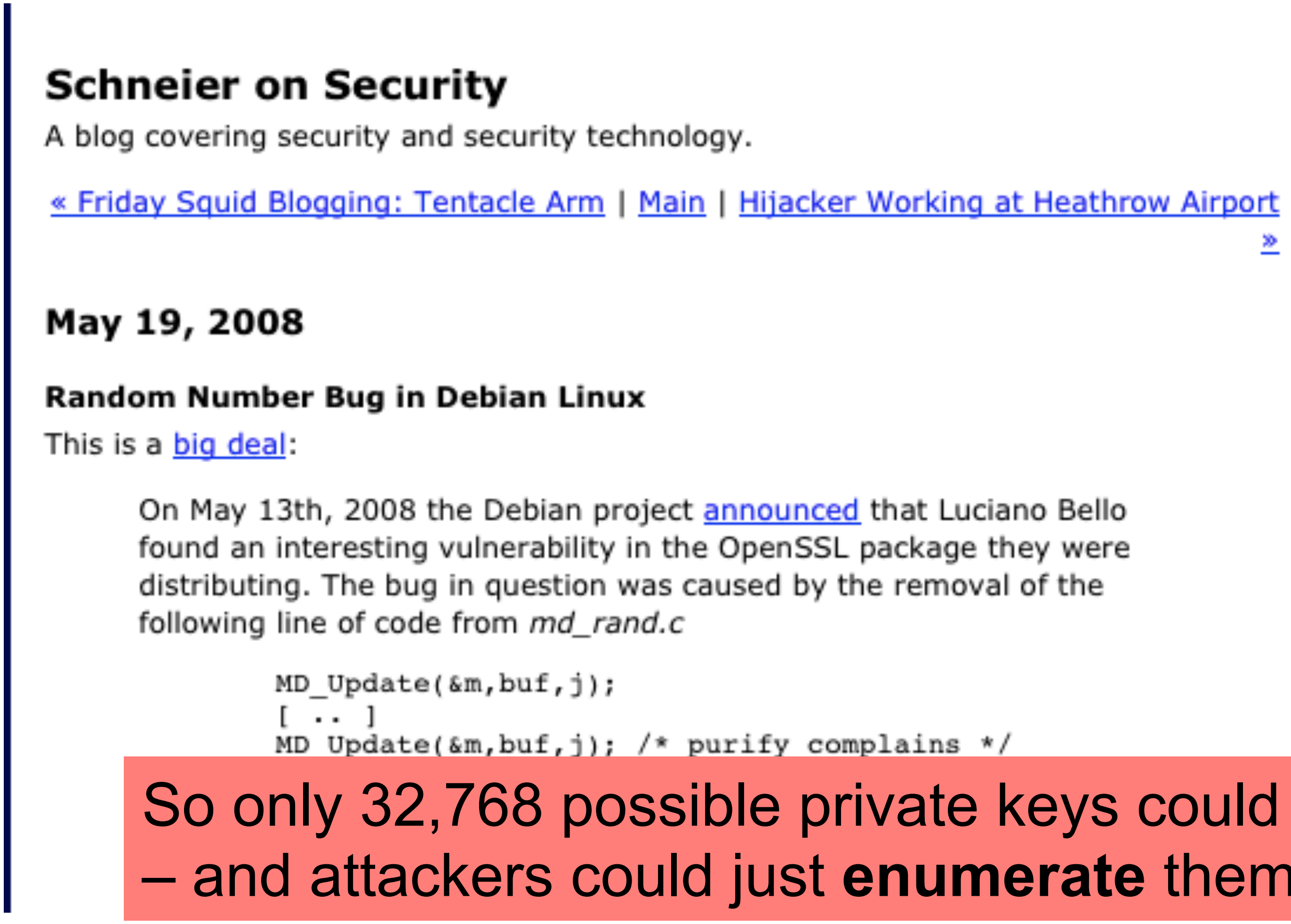
So here's the Apple bug:

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

No demonstration that server possesses private key ⟹ **trivial MITM**

TLS/SSL Trust Issues

- *"Commercial certificate authorities protect you from anyone from whom they are unwilling to take money"*
  - Matt Blaze, circa 2001
- So how many CAs do we have to worry about, anyway?
- Of course, it's not just their greed that matters …
- … and it's not just their diligence & security that matters …
  - *"A decade ago, I observed that commercial certificate authorities protect you from anyone from whom they are unwilling to take money. That turns out to be wrong; they don't even do that much."* - Matt Blaze, circa 2010
- You also have to trust the developers of libraries …
  - Both for clients when validating certs …
  - and servers when generating certs



Schneier on Security

A blog covering security and security technology.

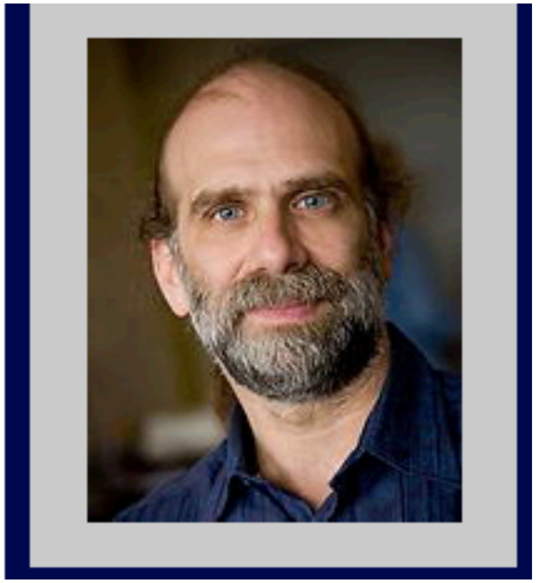
« Friday Squid Blogging: Tentacle Arm | Main | Hijacker Working at Heathrow Airport »

May 19, 2008

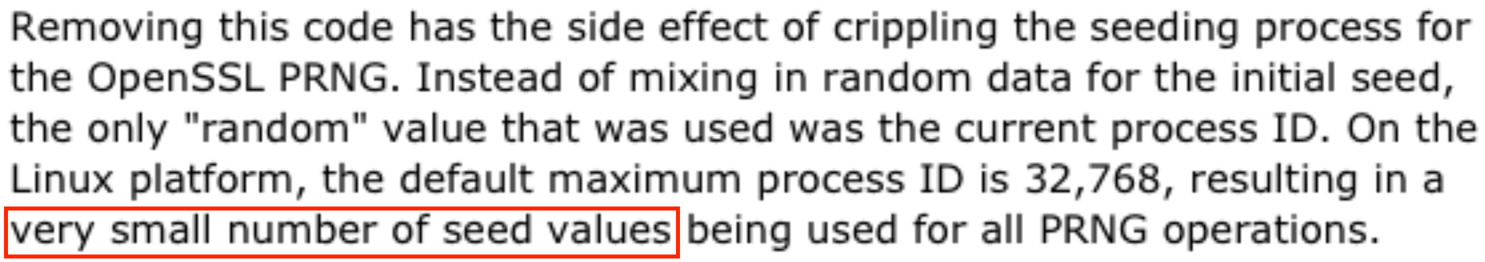
Random Number Bug in Debian Linux

This is a big deal:

> On May 13th, 2008 the Debian project announced that Luciano Bello found an interesting vulnerability in the OpenSSL package they were distributing. The bug in question was caused by the removal of the following line of code from *md_rand.c*
>
> ```
> MD_Update(&m,buf,j);
> [ .. ]
> MD_Update(&m,buf,j); /* purify complains */
> ```



So only 32,768 possible private keys could be generated – and attackers could just **enumerate** them



Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.

Schneier on Security

A blog covering security and security technology.

« Friday Squid Blogging: Tentacle Arm | Main | Hijacker Working at Heathrow Airport »

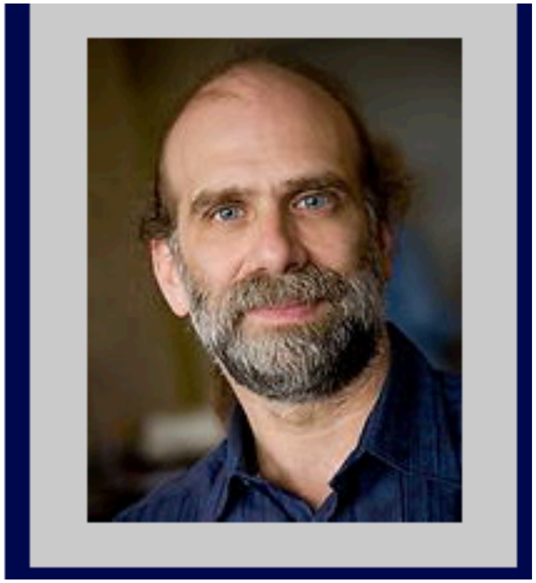
May 19, 2008

Random Number Bug in Debian Linux

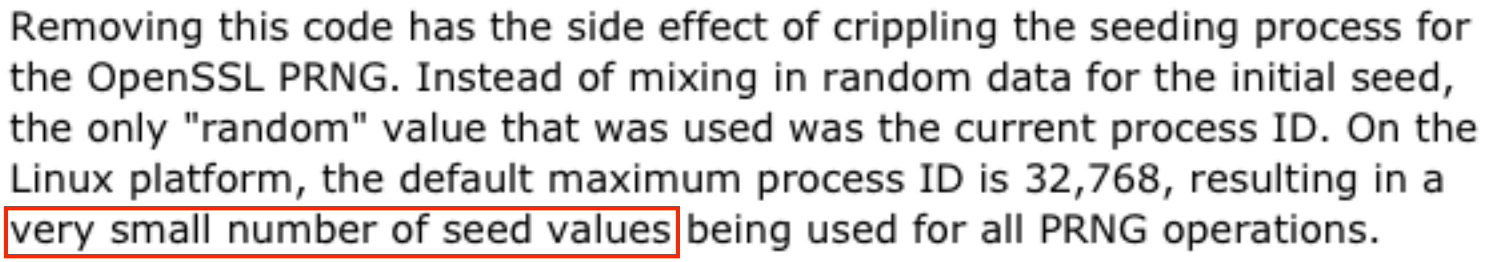
This is a big deal:

> On May 13th, 2008 the Debian project announced that Luciano Bello found an interesting vulnerability in the OpenSSL package they were distributing. The bug in question was caused by the removal of the following line of code from *md_rand.c*

```
MD_Update(&m,buf,j);
[ .. ]
MD_Update(&m,buf,j); /* purify complains */
```



Survey found bug affected ~1.5% of HTTPS web server certs



Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.

5 Minute Break

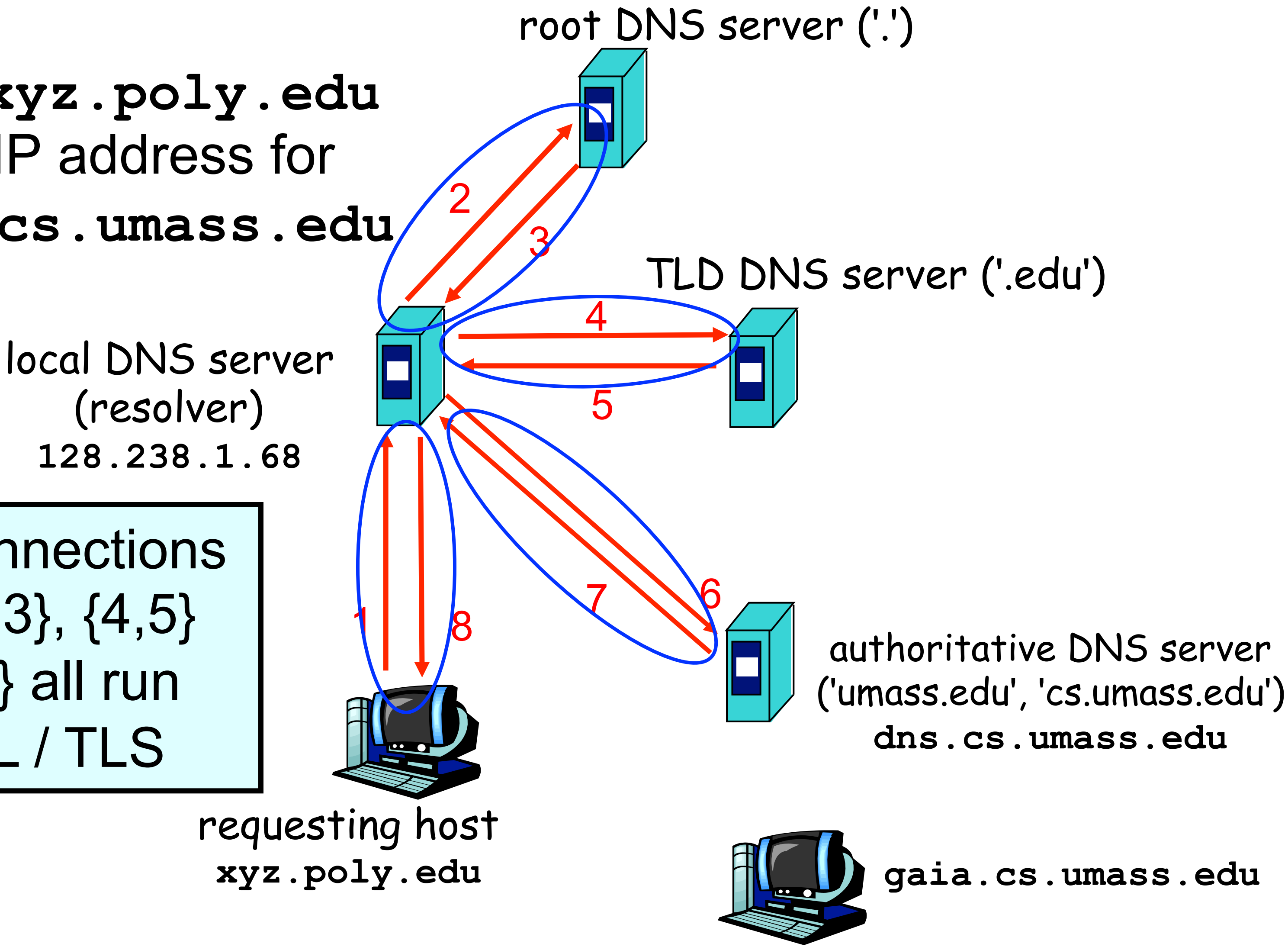
Questions Before We Proceed?

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
`gaia.cs.umass.edu`



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

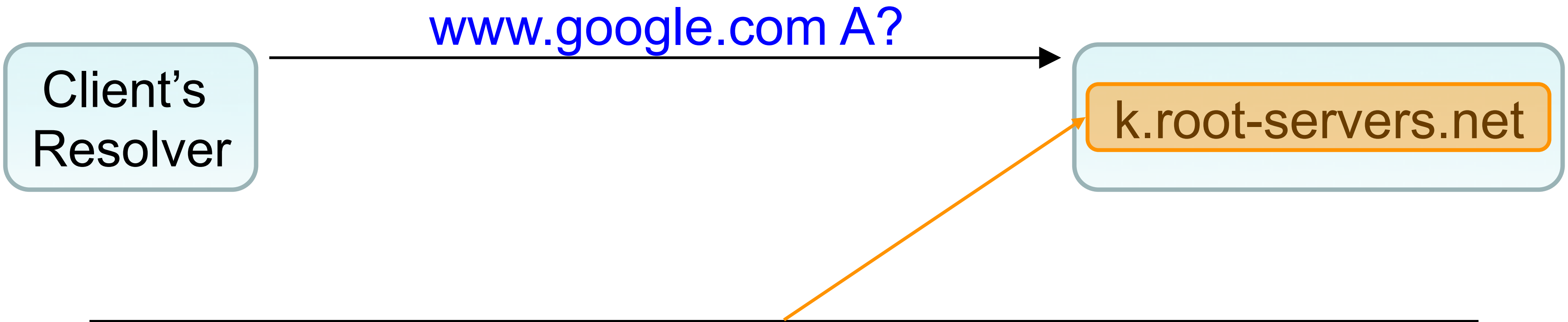
- How can we ensure when clients look up names with DNS, they can trust answers they receive?
- Idea #1: do DNS lookups over TLS
  - (assuming either we run DNS over TCP, or we use "Datagram TLS")
  - Issues?
    - Performance: DNS is very lightweight. TLS is not.
    - Caching: crucial for DNS scaling.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

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*

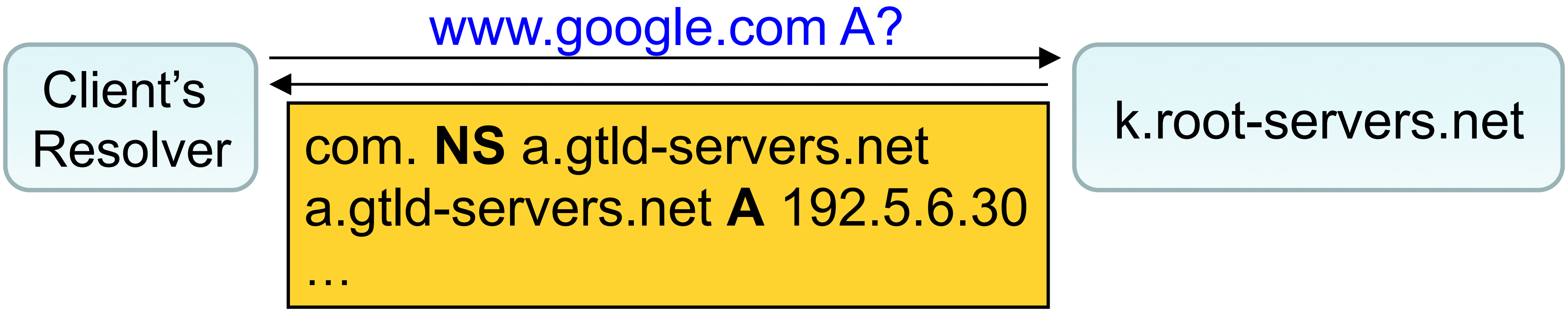
Ordinary DNS:

Ordinary DNS:



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:



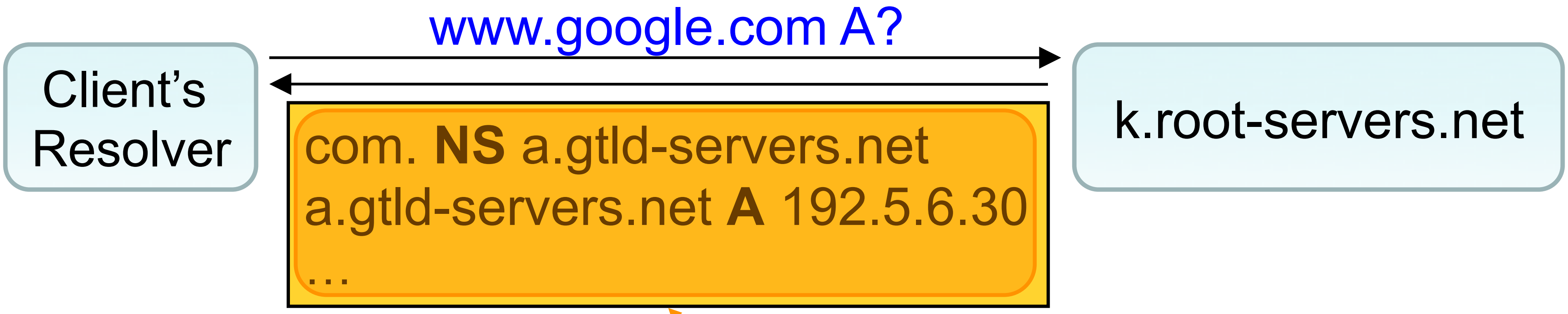
Client's Resolver

www.google.com A?

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net **A** 192.5.6.30
…

Ordinary DNS:



Client's Resolver

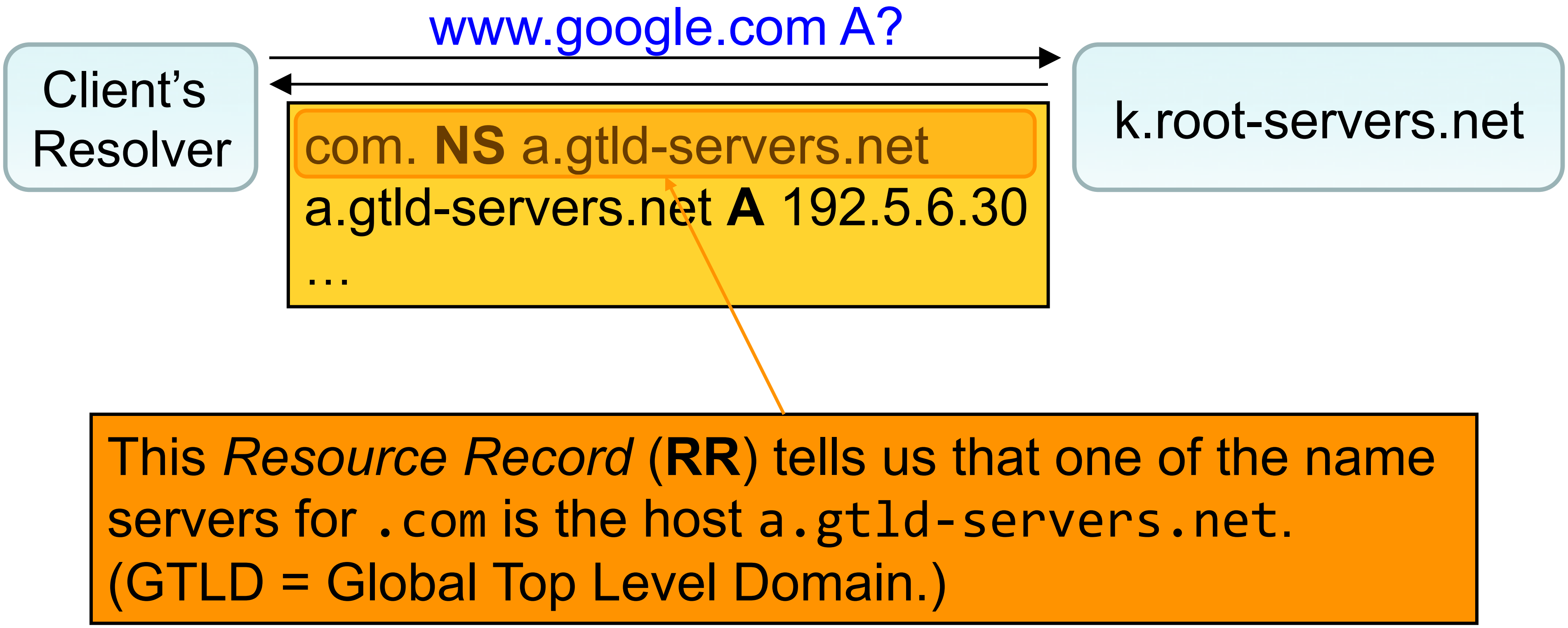
www.google.com A?

k.root-servers.net

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

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:



Client's Resolver

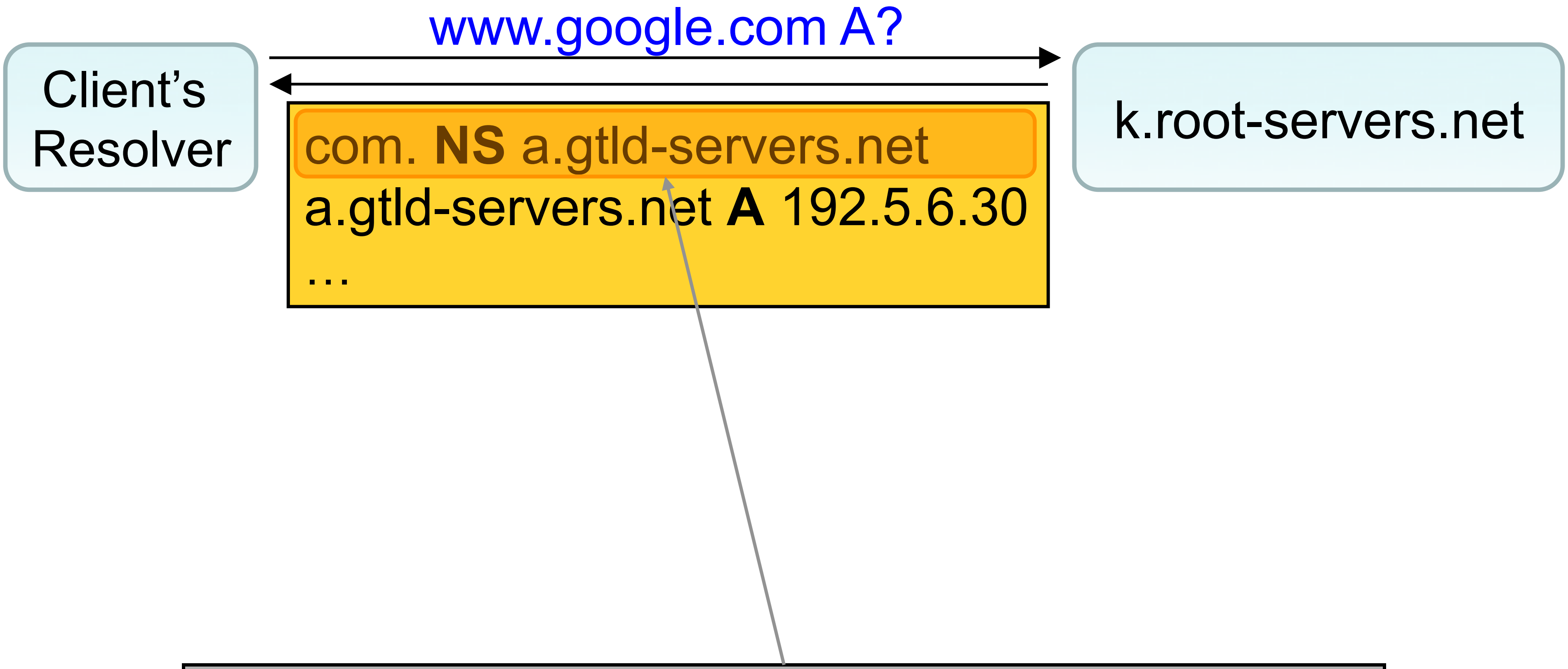
www.google.com A?

k.root-servers.net

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

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:



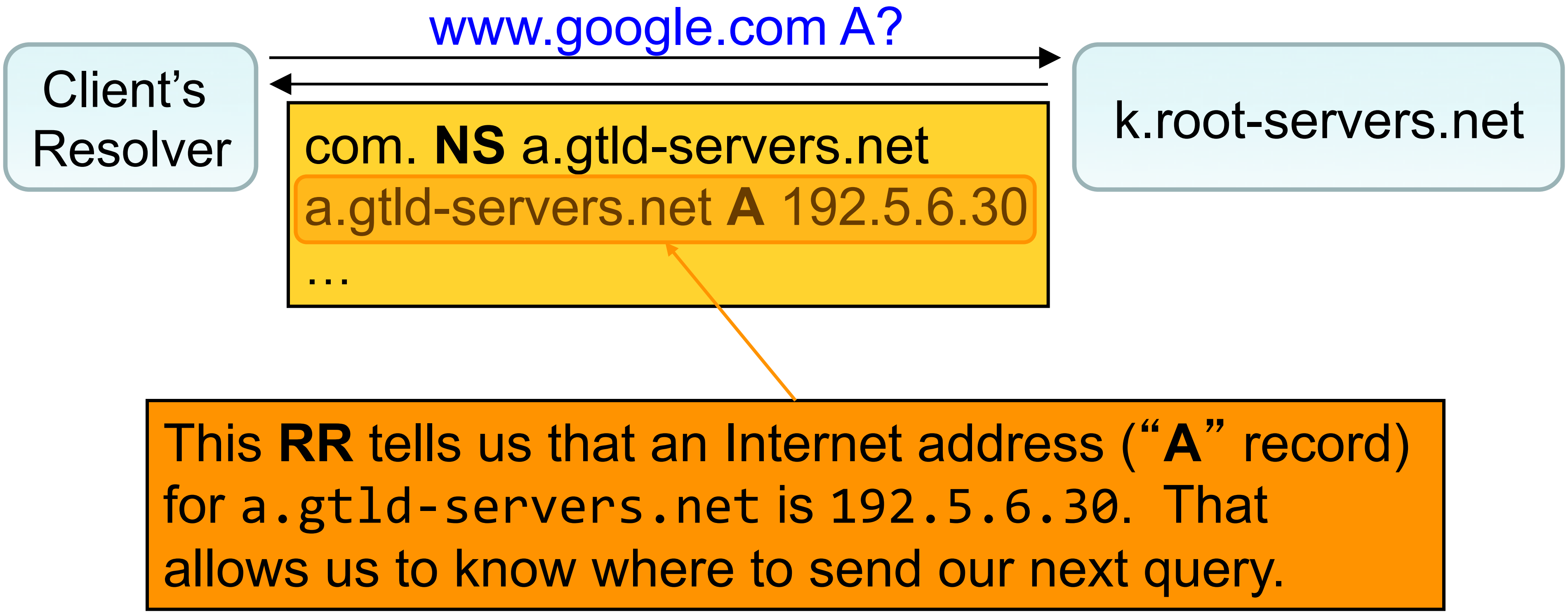
com. **NS** a.gtld-servers.net

a.gtld-servers.net **A** 192.5.6.30

...

(The line above shows `com.` rather than `.com` because technically that's the actual name, and that's what the Unix `dig` utility shows; but the convention is to *call* it "dot-com")

Ordinary DNS:



Client's Resolver

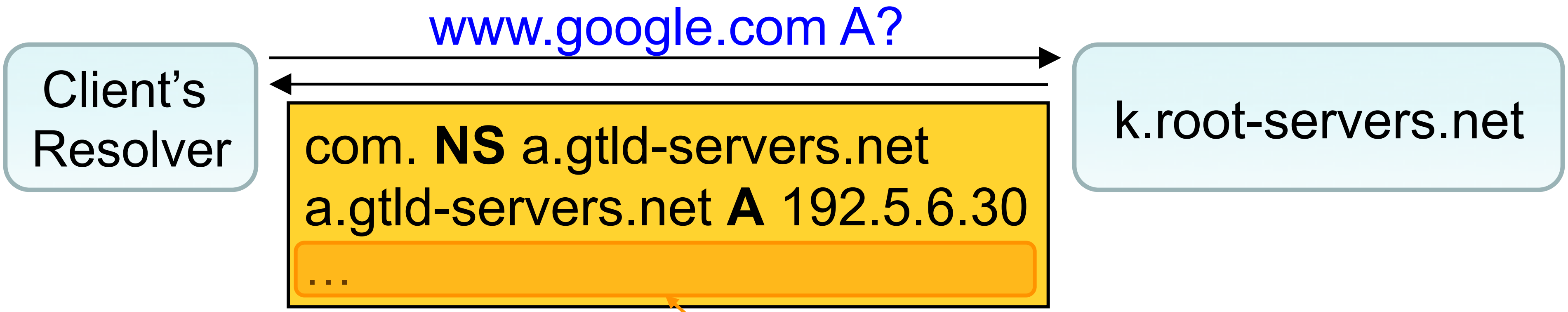
www.google.com A?

k.root-servers.net

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

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:



Client's Resolver

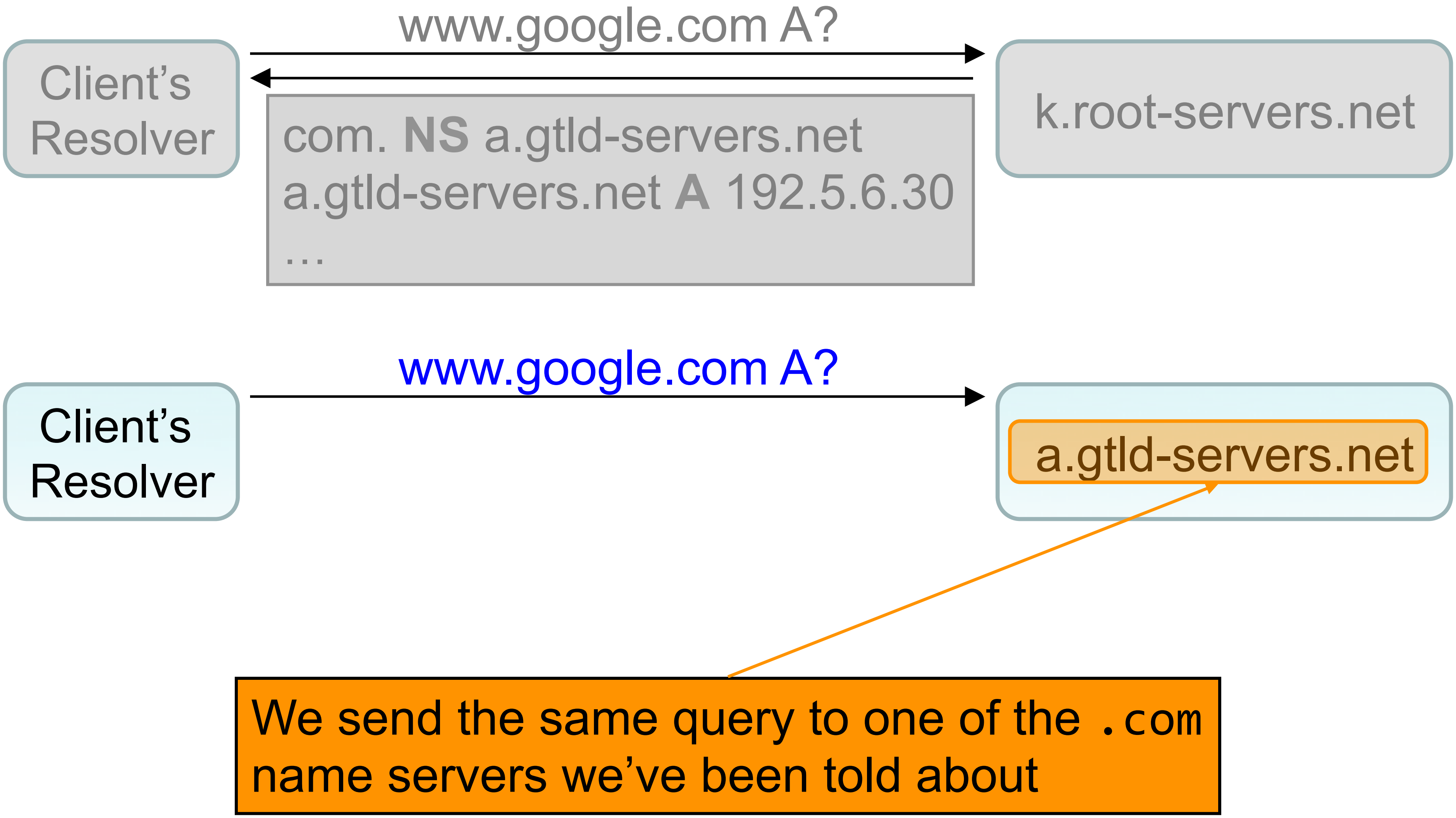
www.google.com A?

k.root-servers.net

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

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

Ordinary DNS:



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
…

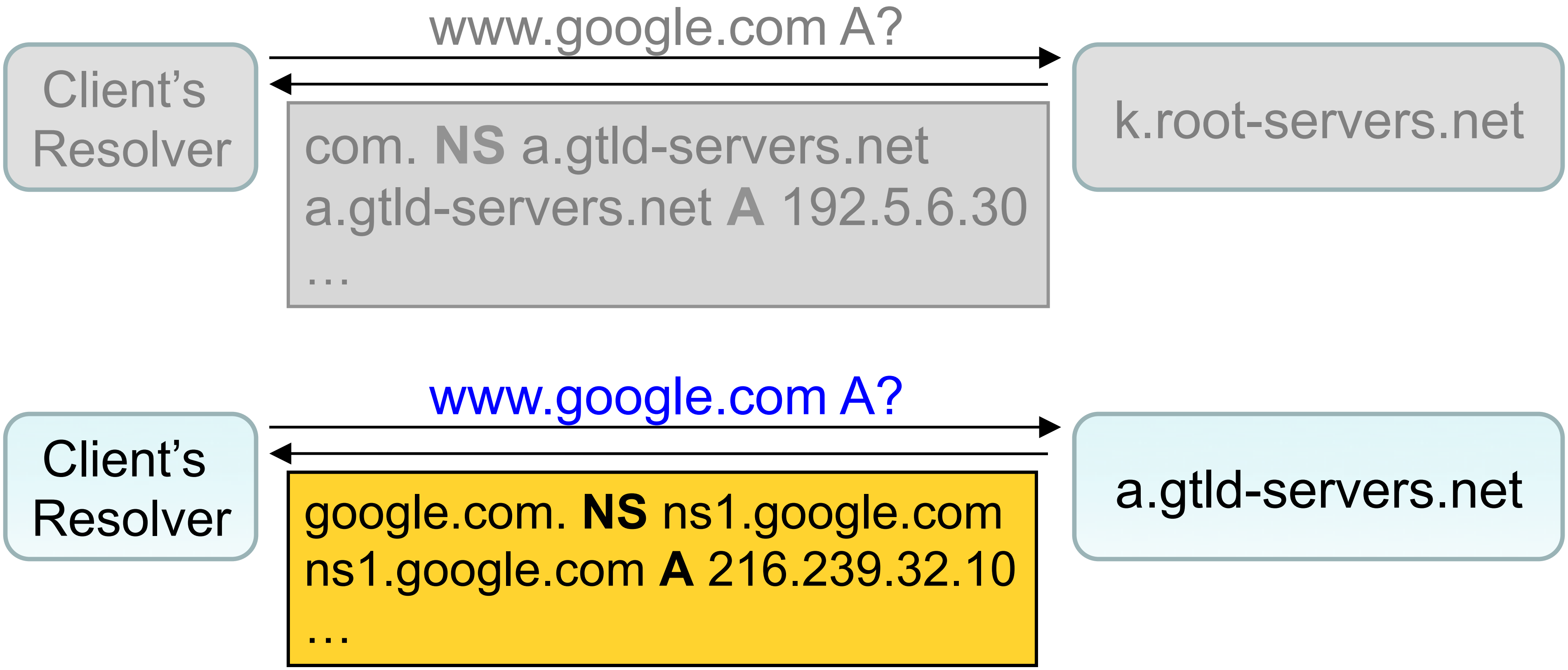
Client's Resolver

www.google.com A?

a.gtld-servers.net

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

Ordinary DNS:



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
…

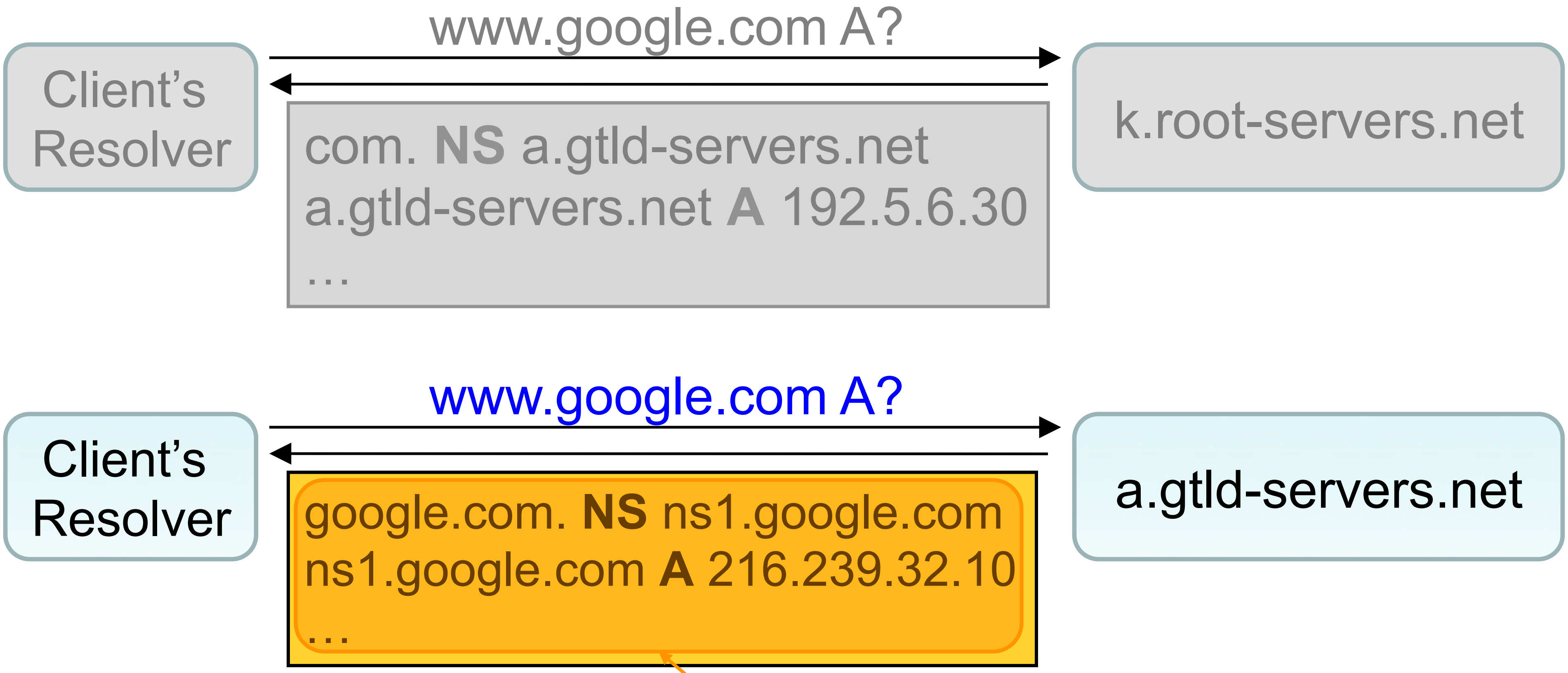
Client's Resolver

www.google.com A?

a.gtld-servers.net

google.com. **NS** ns1.google.com
ns1.google.com **A** 216.239.32.10
…

Ordinary DNS:



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
. . .

Client's Resolver

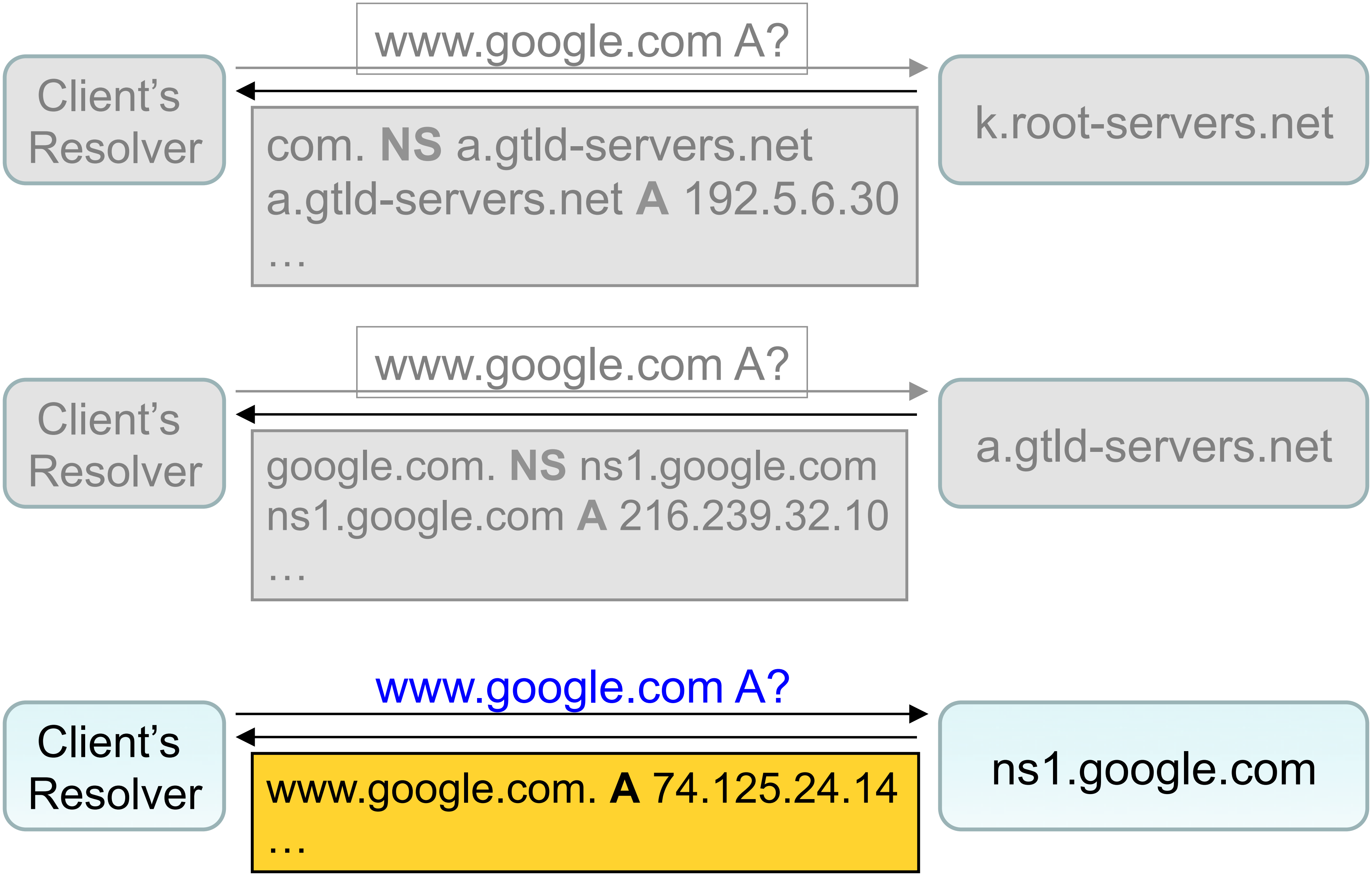
www.google.com A?

a.gtld-servers.net

google.com. **NS** ns1.google.com
ns1.google.com **A** 216.239.32.10
. . .

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

Ordinary DNS:



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
…

Client's Resolver

www.google.com A?

a.gtld-servers.net

google.com. **NS** ns1.google.com
ns1.google.com **A** 216.239.32.10
…

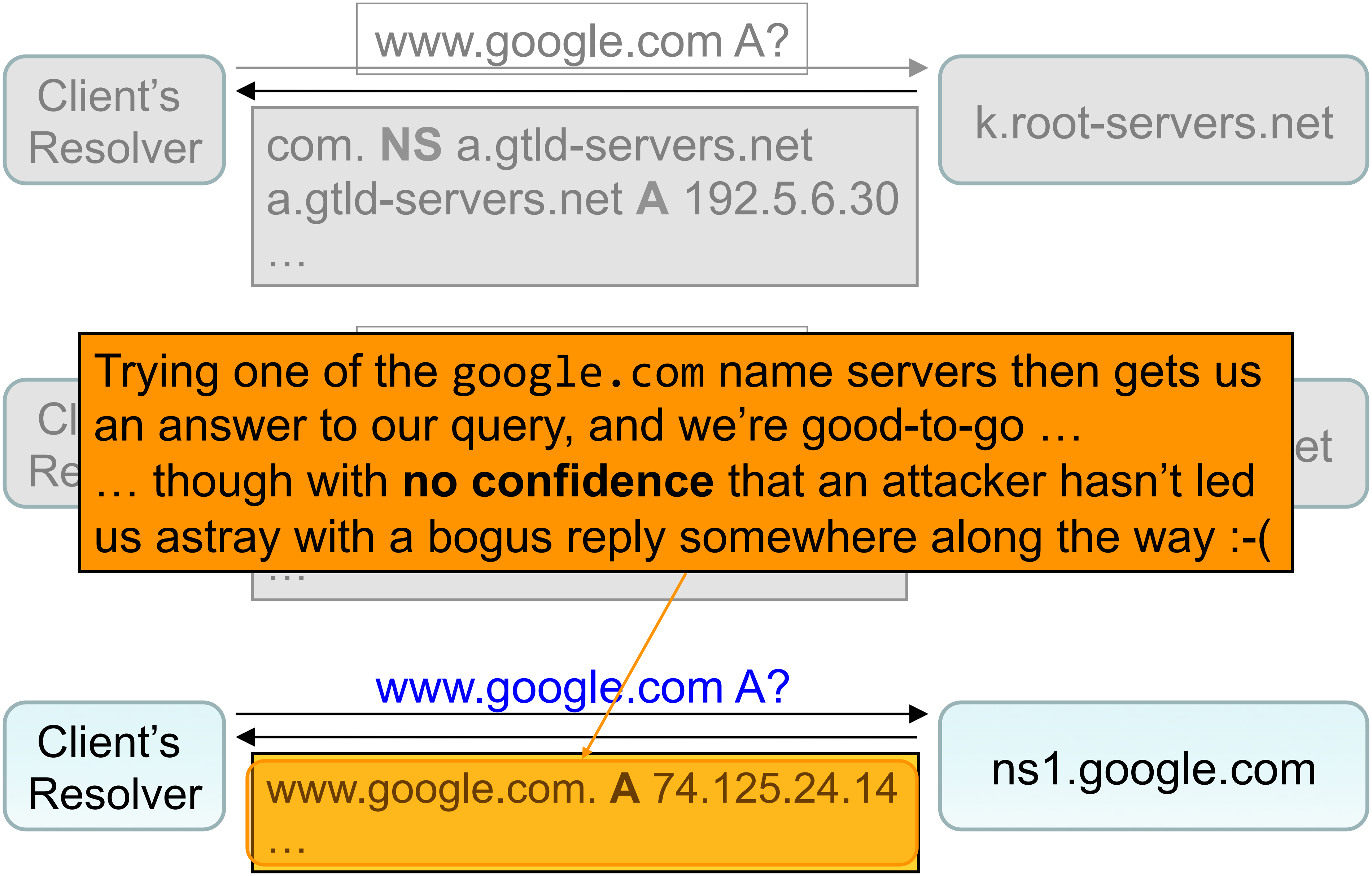
Client's Resolver

www.google.com A?

ns1.google.com

www.google.com. **A** 74.125.24.14
…

Ordinary DNS:



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
. . .

Cl
Re

et

. . .

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 :-(

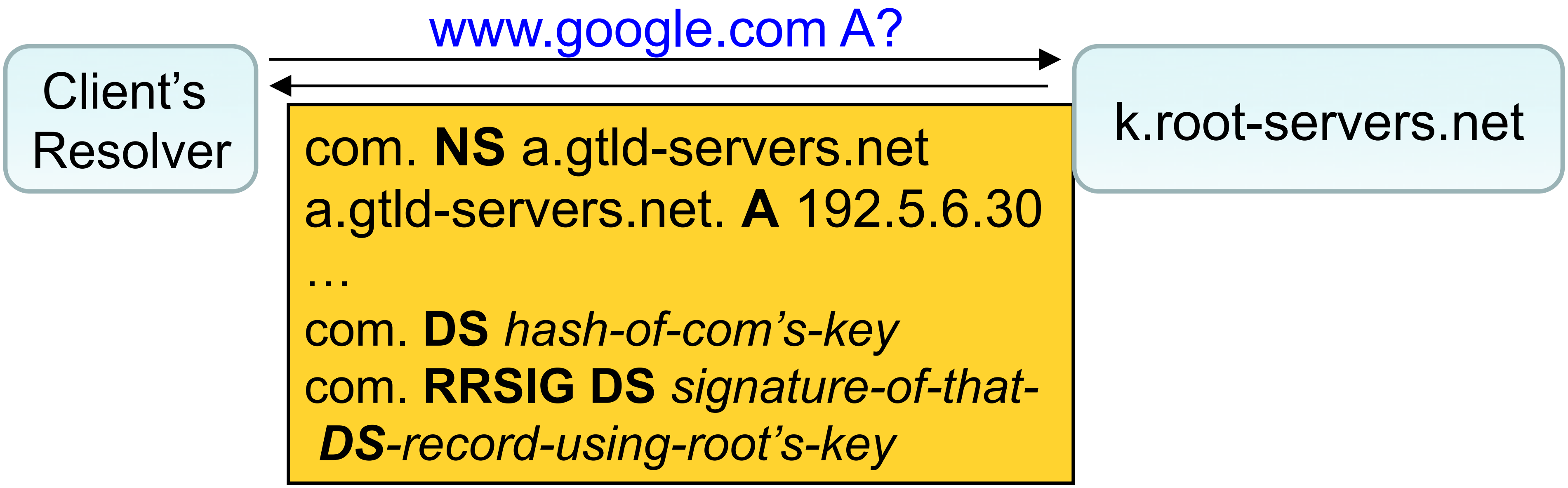
Client's Resolver

www.google.com A?

ns1.google.com

www.google.com. **A** 74.125.24.14
. . .

DNSSEC (with simplifications):



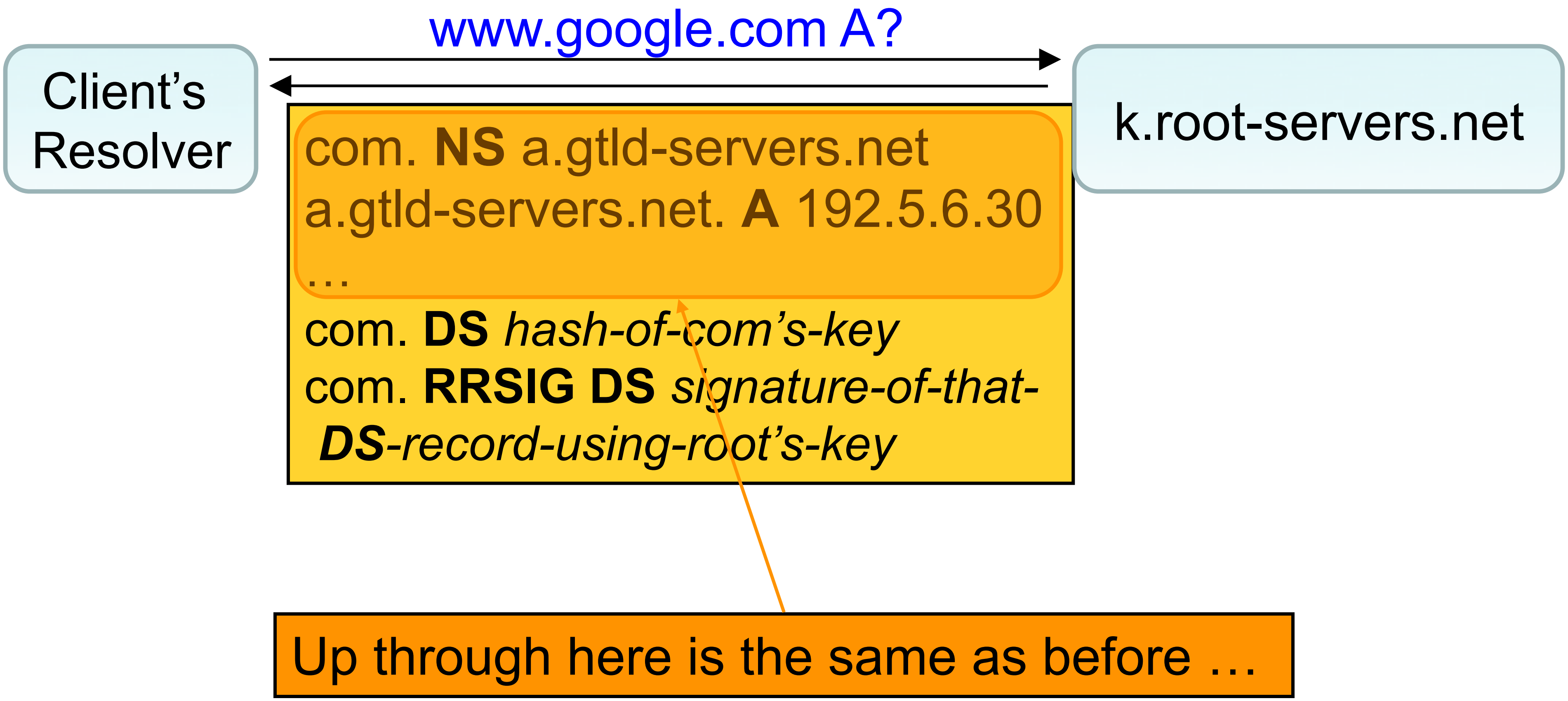
Client's Resolver

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*

DNSSEC (with simplifications):



Client's Resolver

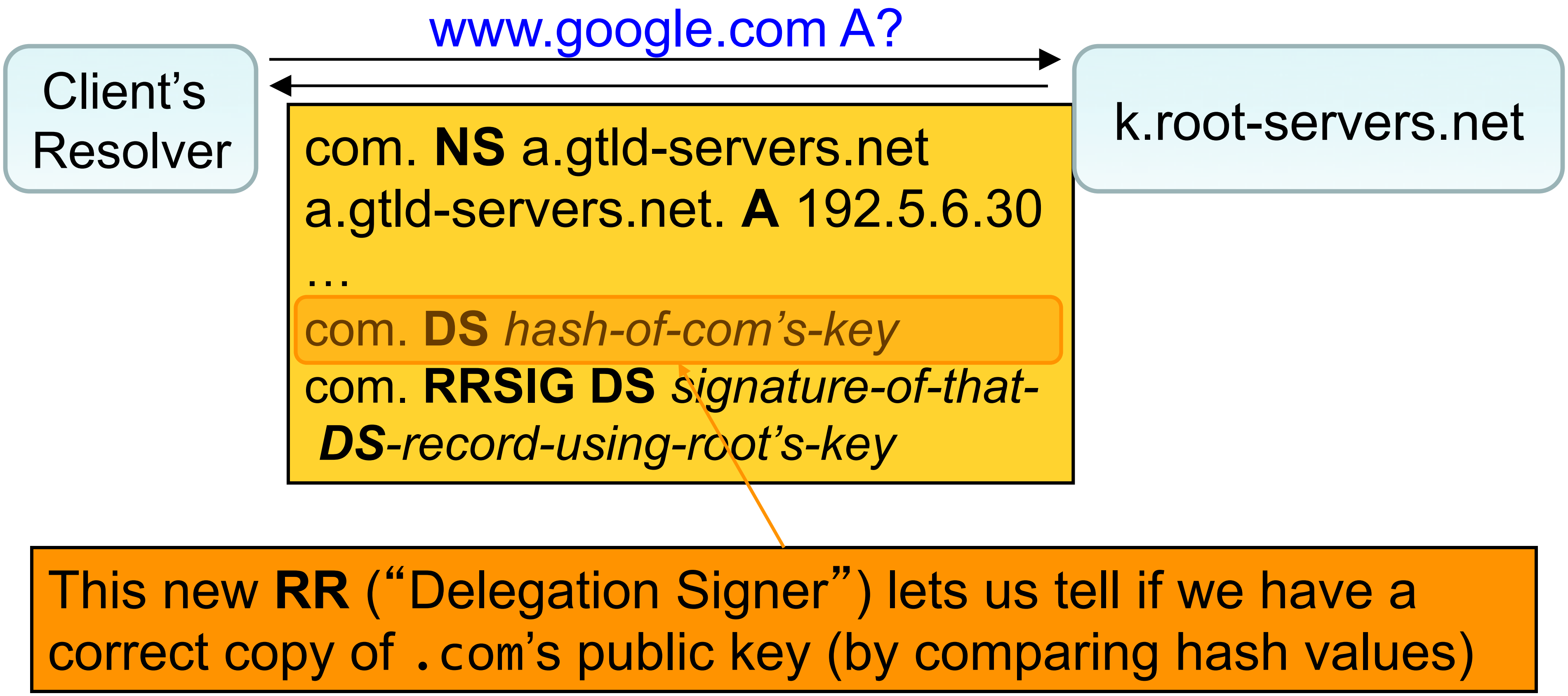
www.google.com A?

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*

Up through here is the same as before …

DNSSEC (with simplifications):



Client's Resolver

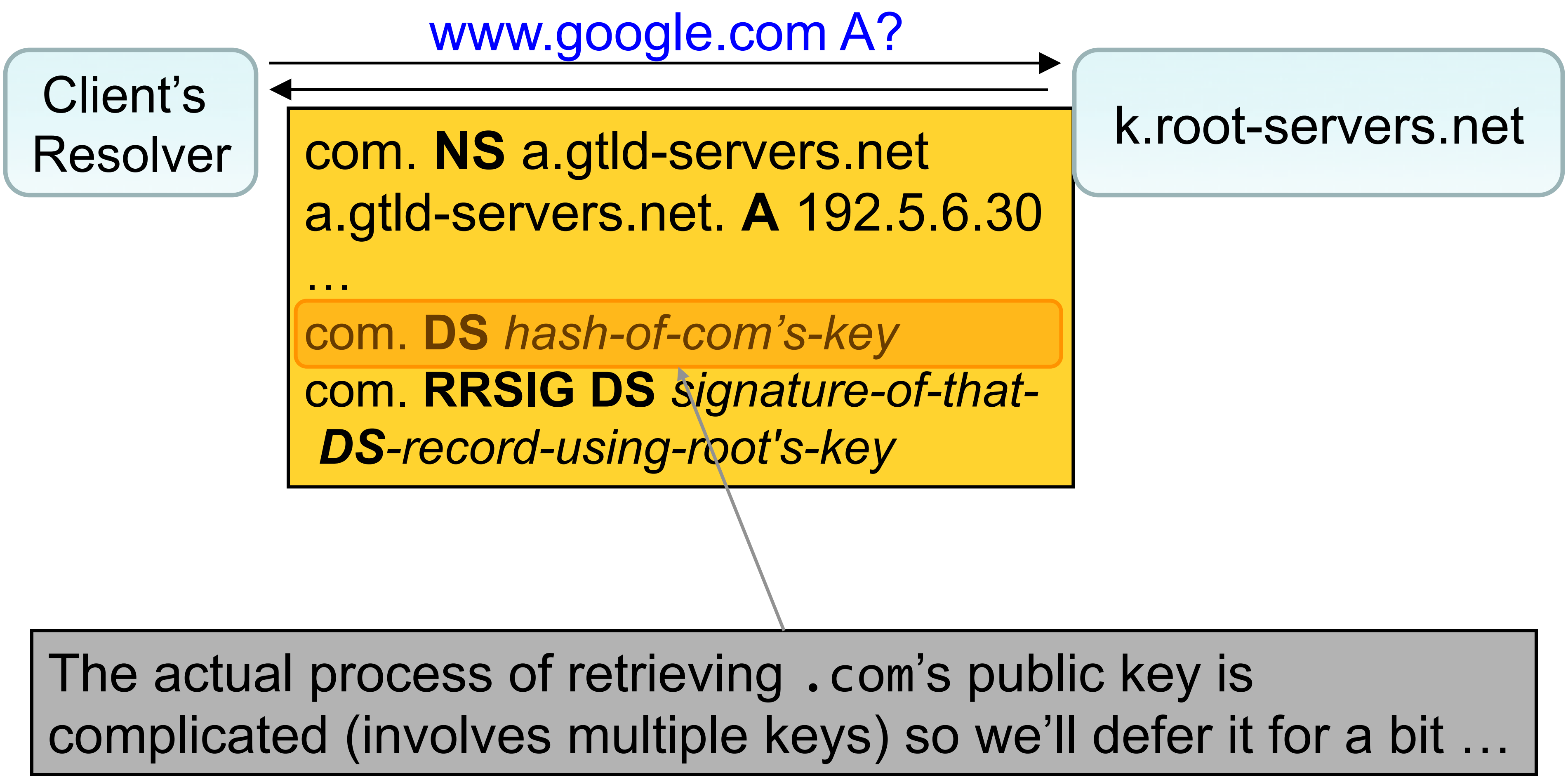
www.google.com A?

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*

This new **RR** ("Delegation Signer") lets us tell if we have a correct copy of `.com`'s public key (by comparing hash values)

DNSSEC (with simplifications):



www.google.com A?

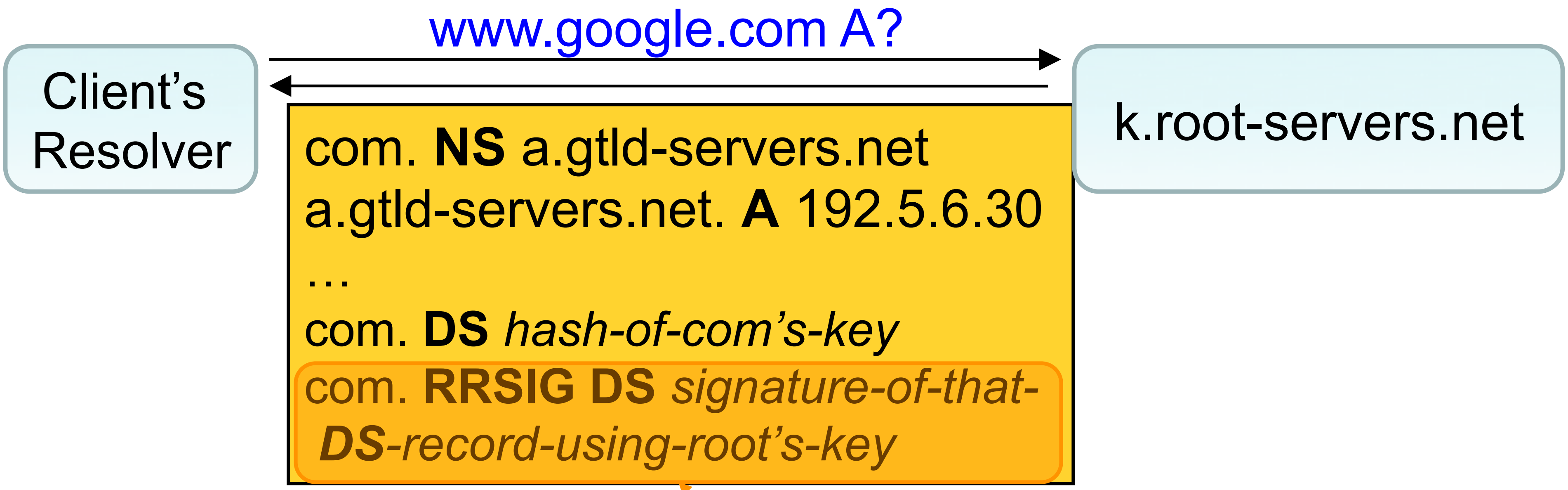
Client's Resolver

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*

The actual process of retrieving `.com`'s public key is complicated (involves multiple keys) so we'll defer it for a bit …

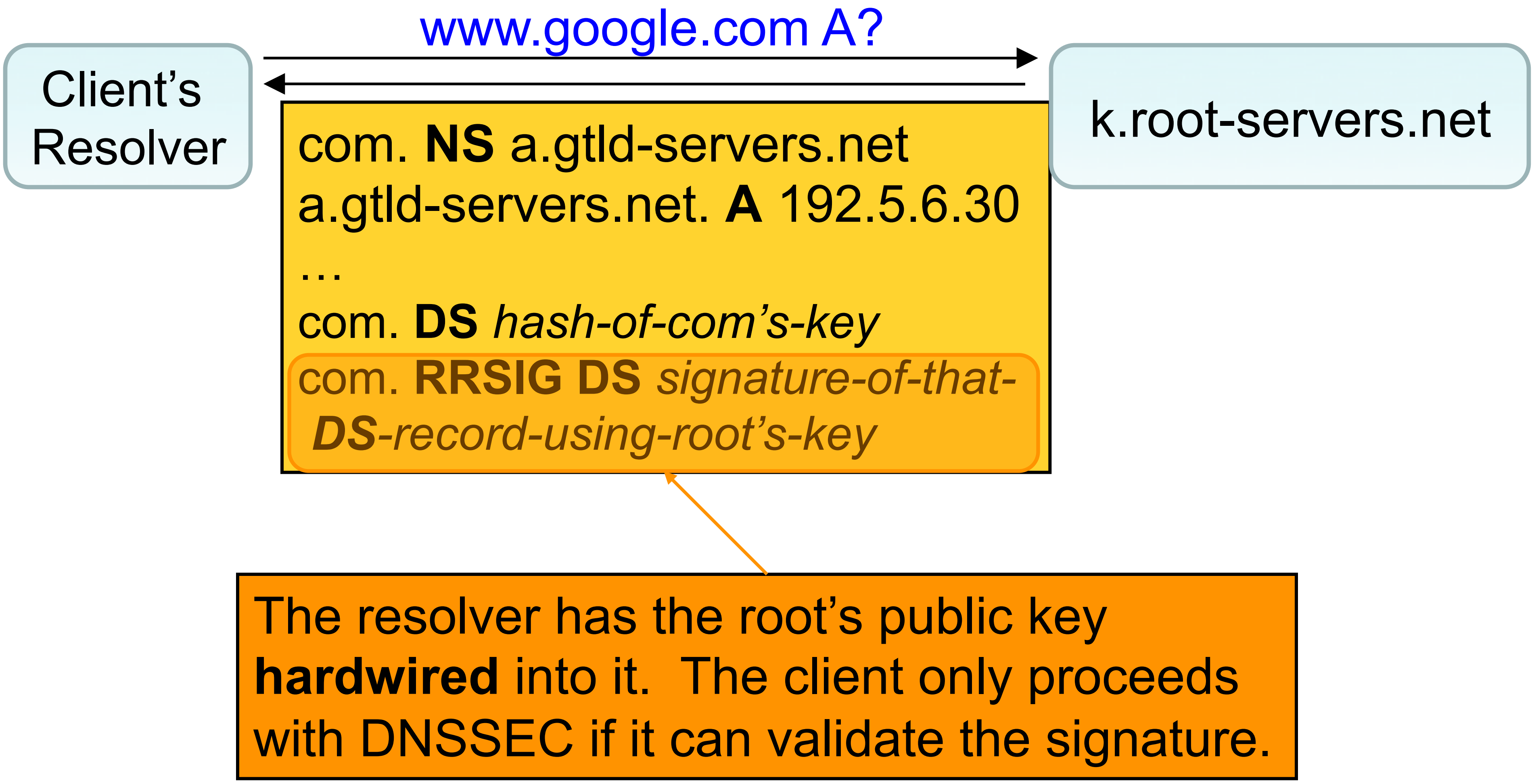
DNSSEC (with simplifications):



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*

This new **RR** specifies a <u>signature</u> 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):



Client's Resolver

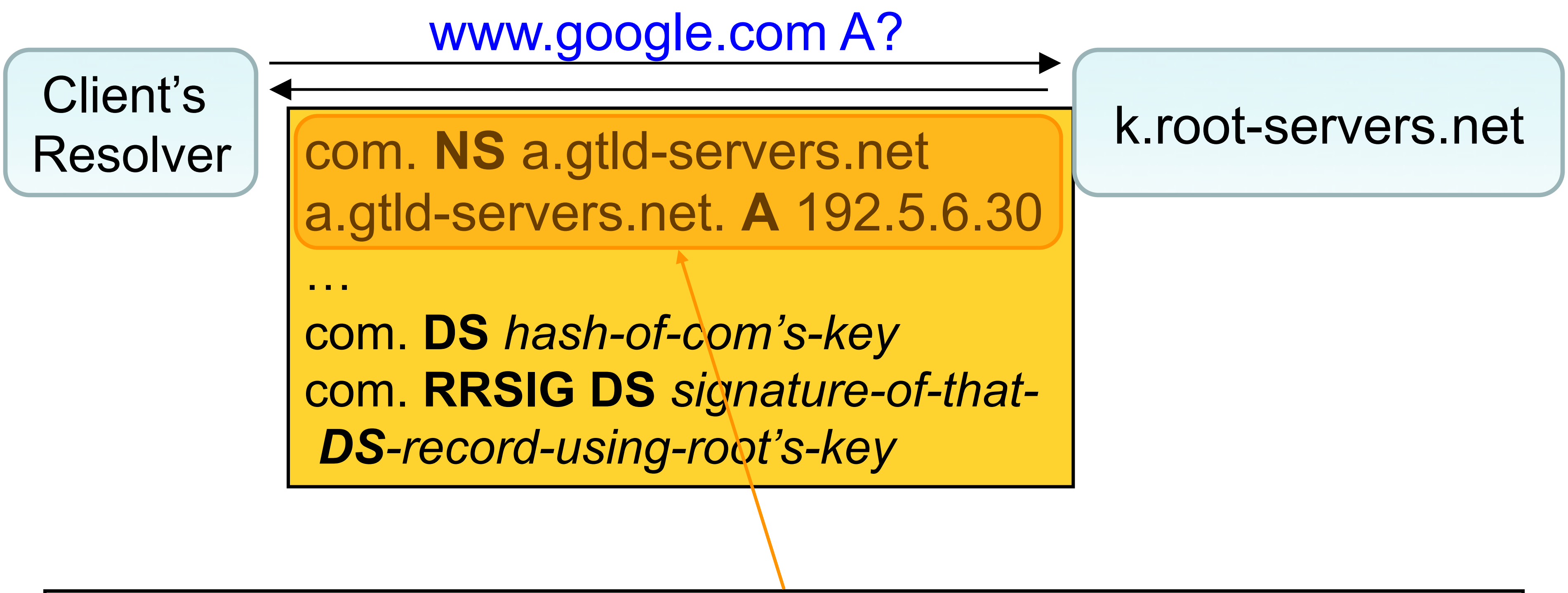
www.google.com A?

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*

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):



Client's Resolver

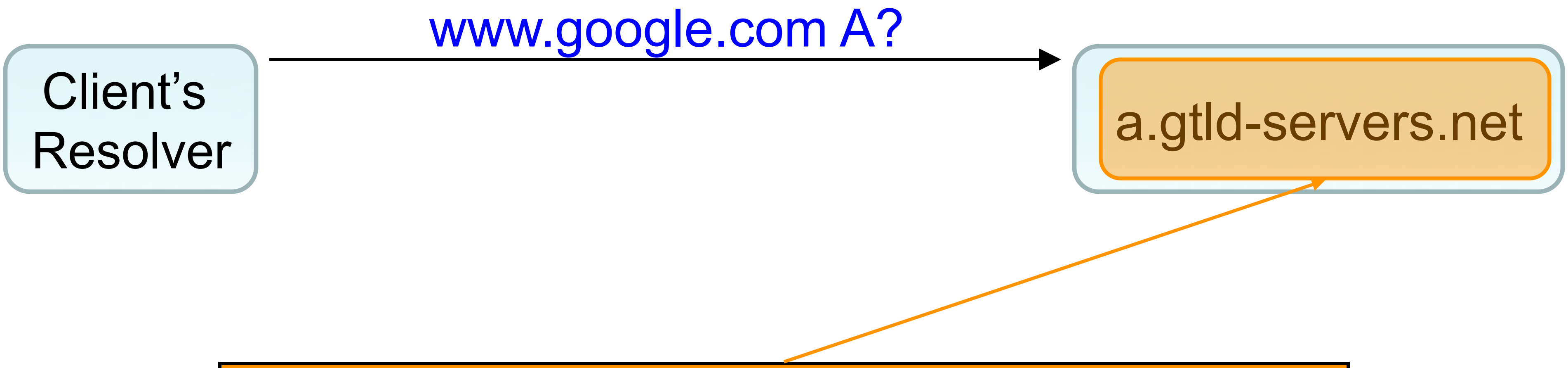
www.google.com A?

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*

Note: there's no signature over the **NS** or **A** information! If an attacker has fiddled with those, the resolver will ultimately find it has a record for which it can't verify the signature.

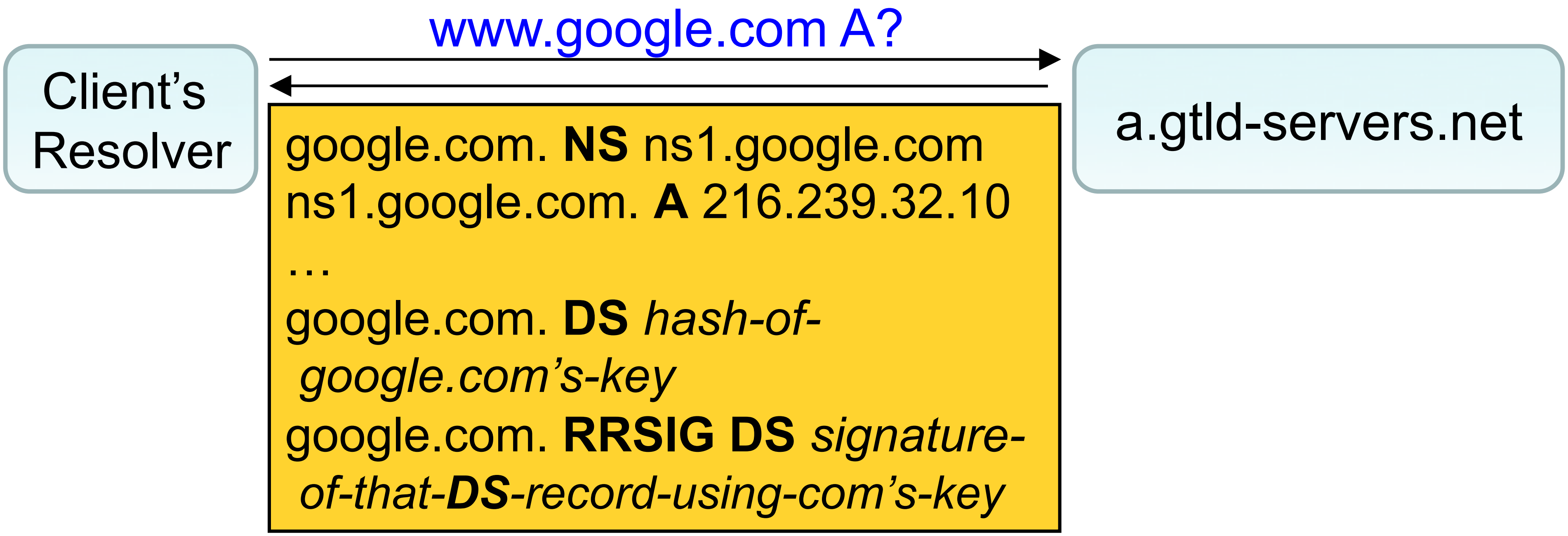
DNSSEC (with simplifications):



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):



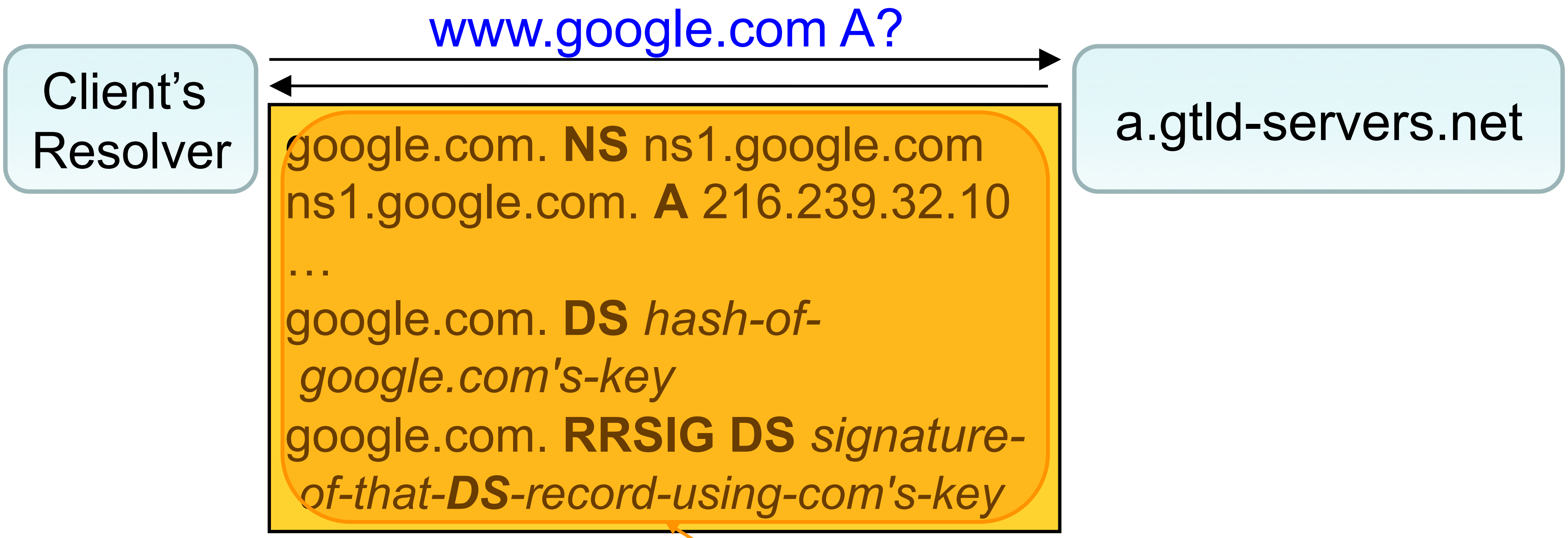
Client's Resolver

www.google.com A?

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*

DNSSEC (with simplifications):



Client’s Resolver

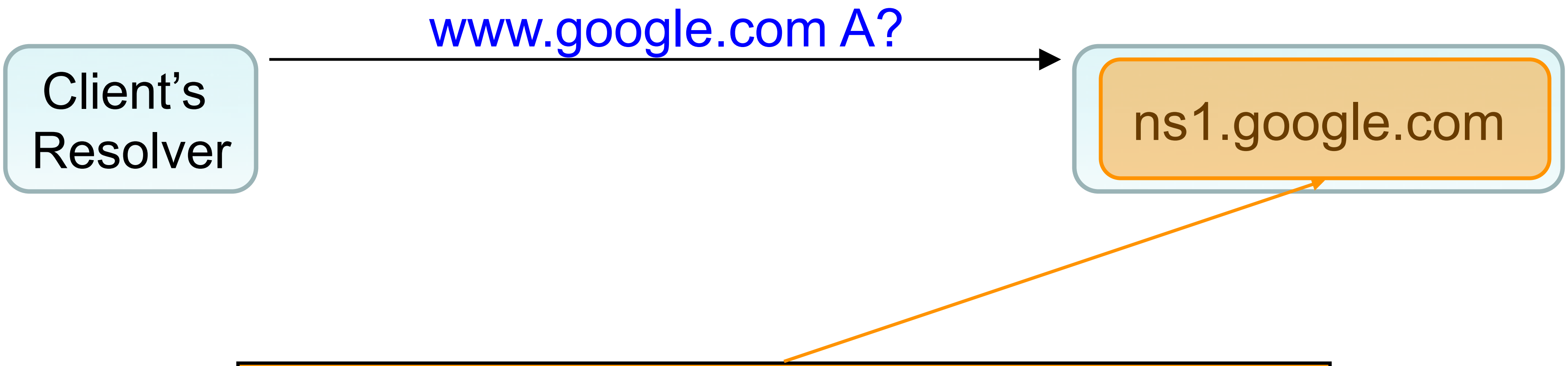
www.google.com A?

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*

Back comes similar information as before: a way to securely identify `google.com`’s public key, signed by `.com`’s key (which the resolver trusts because the root signed information about it)

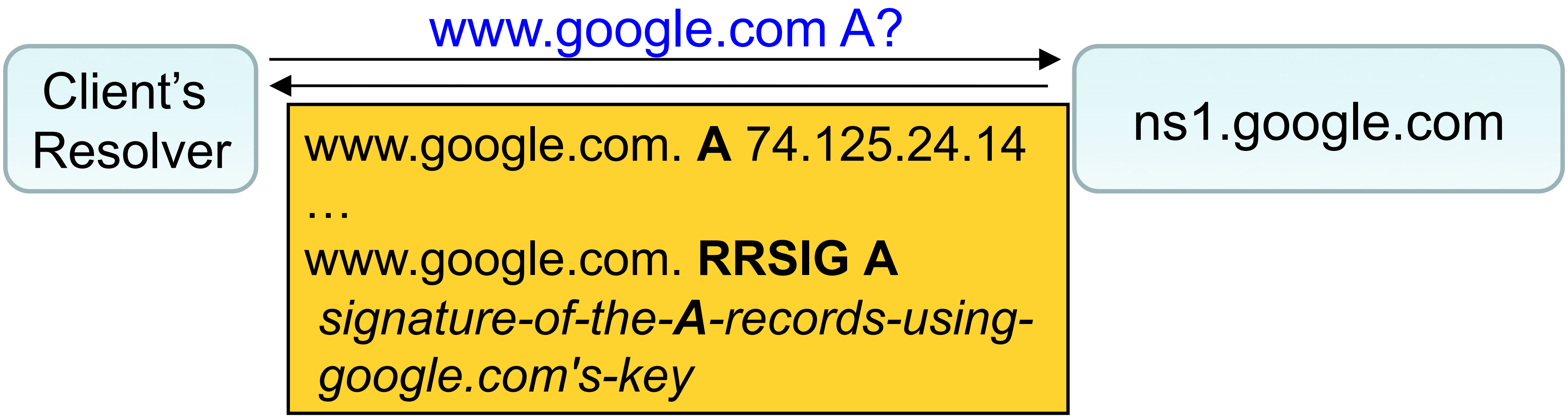
DNSSEC (with simplifications):



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):



Client's Resolver

www.google.com A?

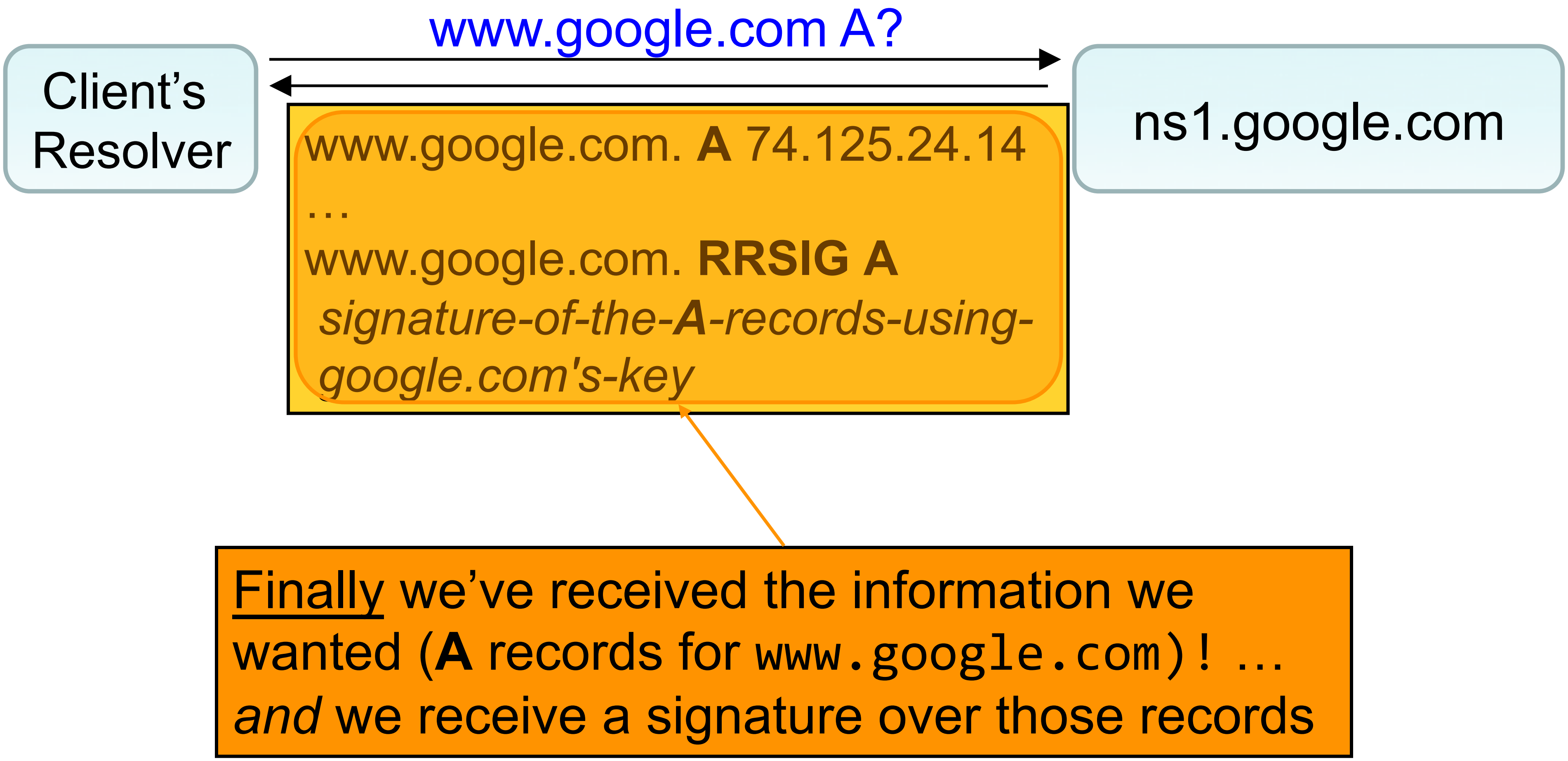
ns1.google.com

www.google.com. **A** 74.125.24.14

…

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

DNSSEC (with simplifications):



Client's Resolver

www.google.com A?

ns1.google.com

www.google.com. **A** 74.125.24.14

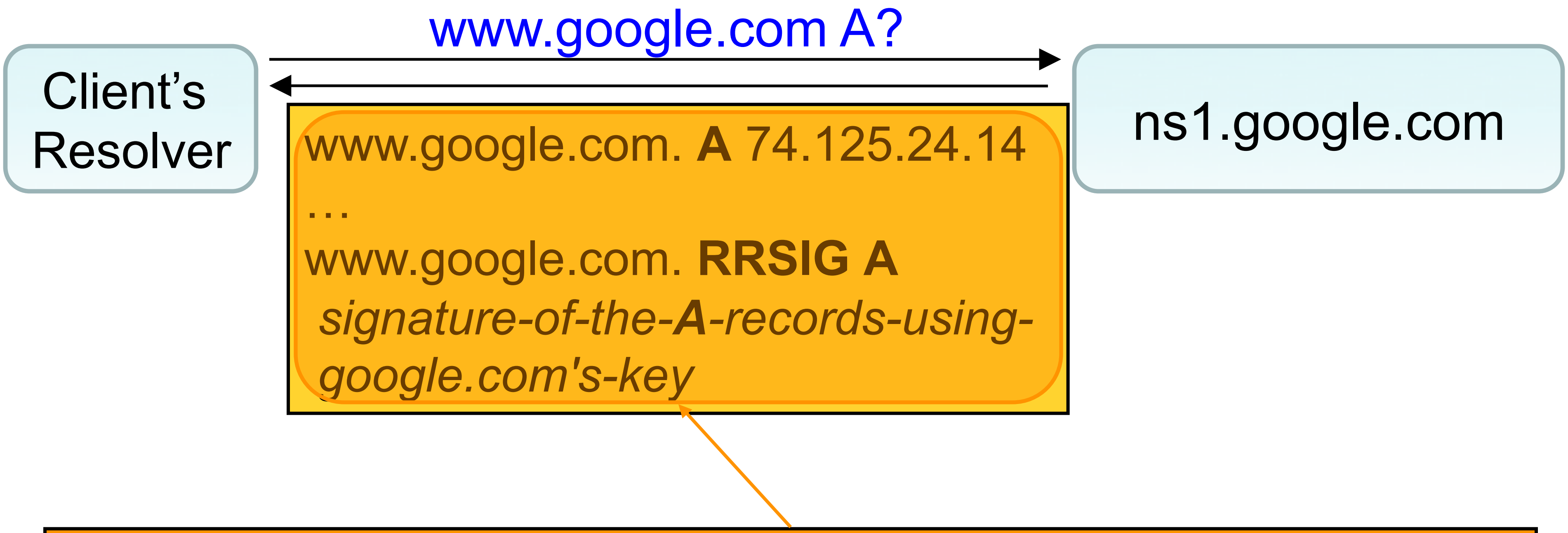
…

www.google.com. **RRSIG A**

*signature-of-the-**A**-records-using-google.com's-key*

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):



Client's Resolver

www.google.com A?

ns1.google.com

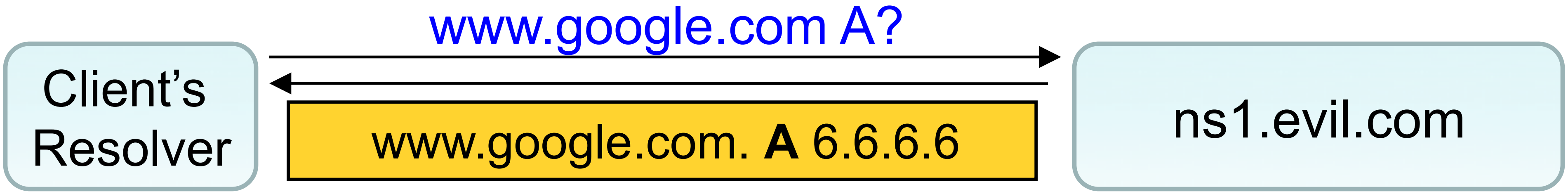
www.google.com. **A** 74.125.24.14

…

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

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!



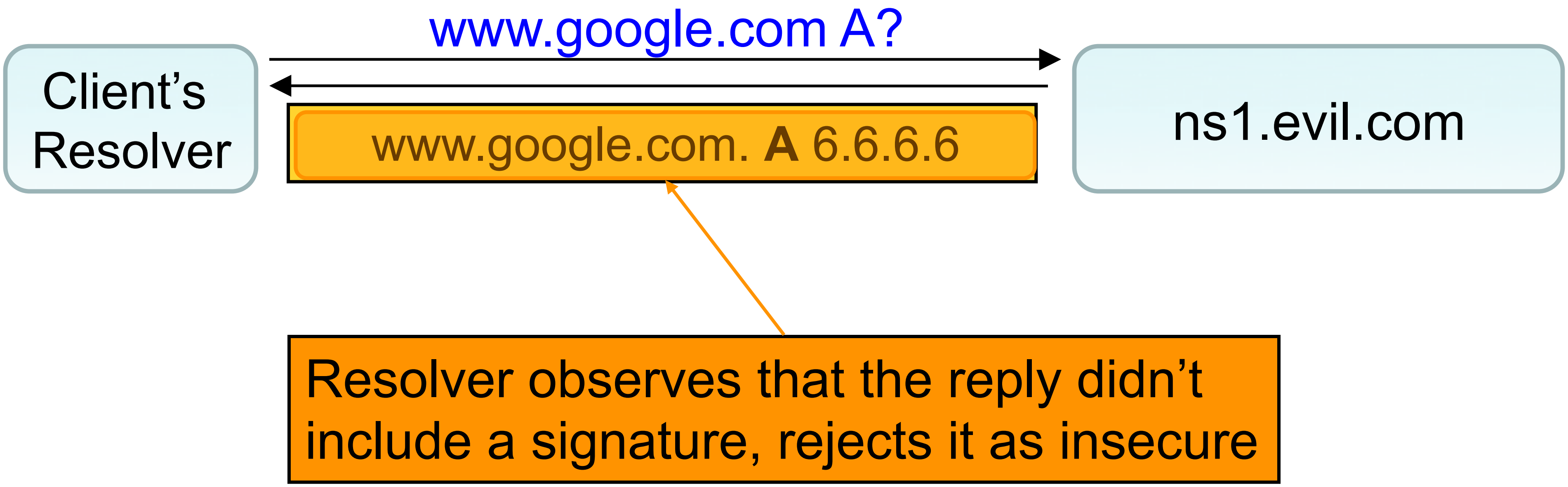
Client's Resolver

www.google.com A?

www.google.com. **A** 6.6.6.6

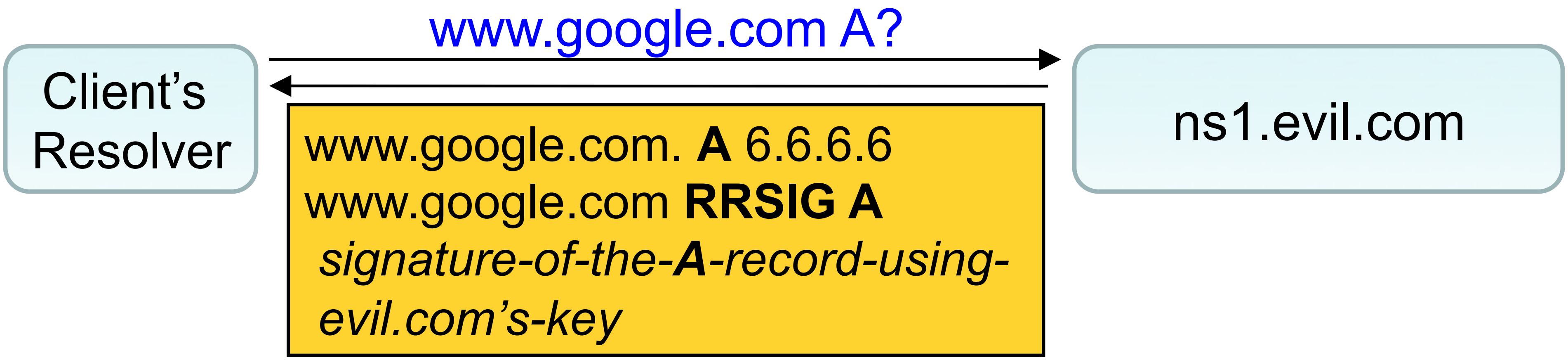
ns1.evil.com

DNSSEC - Mallory attacks!



Resolver observes that the reply didn't include a signature, rejects it as insecure

DNSSEC - Mallory attacks!



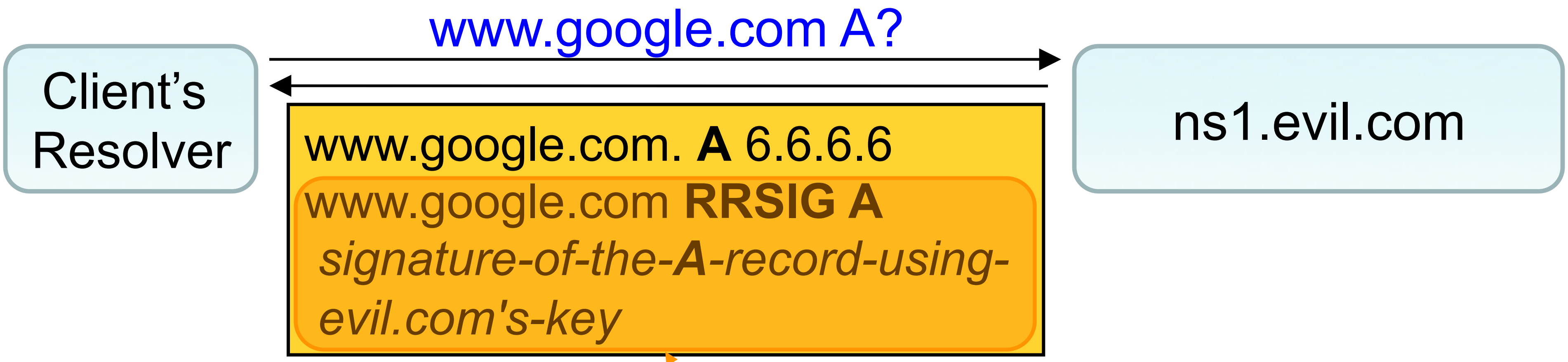
Client's Resolver

www.google.com A?

ns1.evil.com

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

DNSSEC - Mallory attacks!



Client's Resolver

www.google.com A?

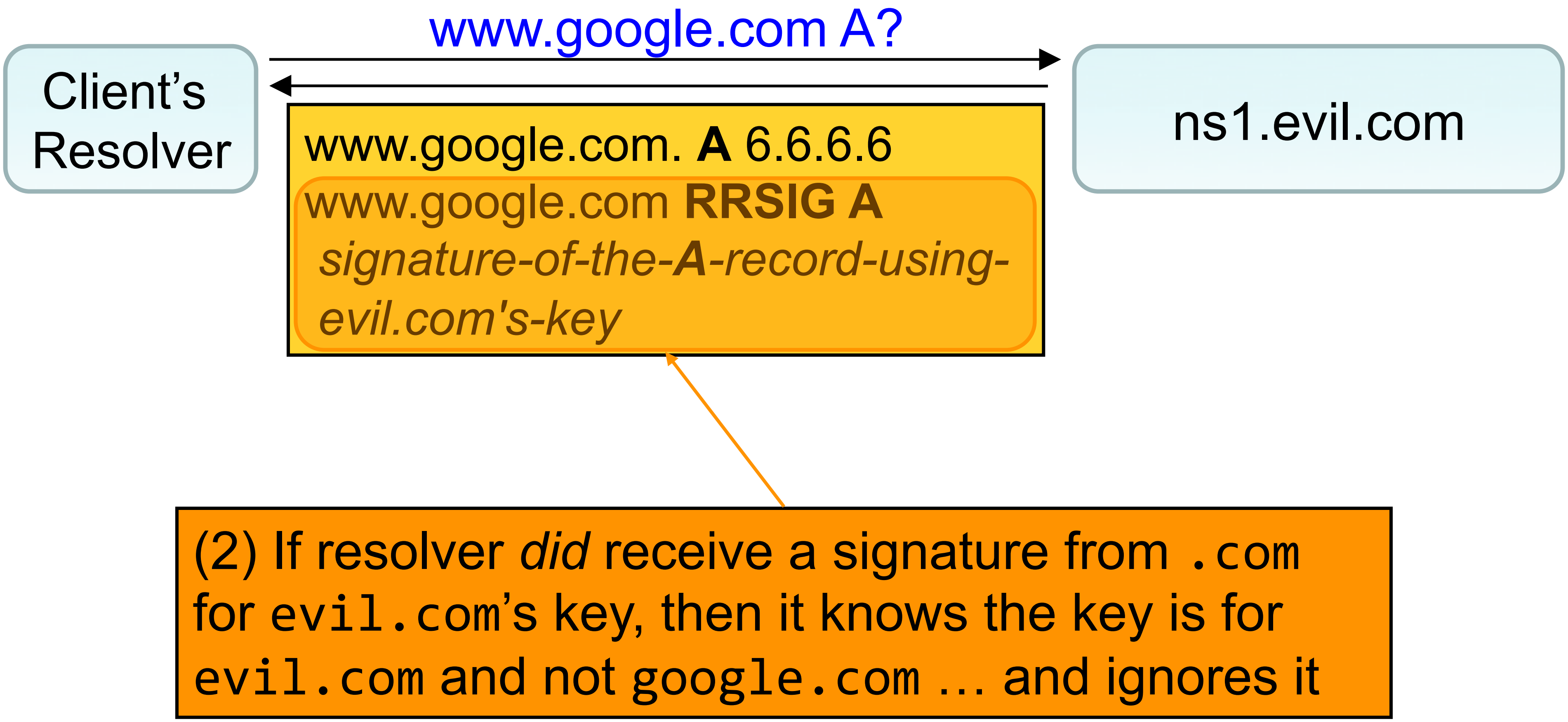
ns1.evil.com

www.google.com. **A** 6.6.6.6

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

(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!



Client's Resolver

www.google.com A?

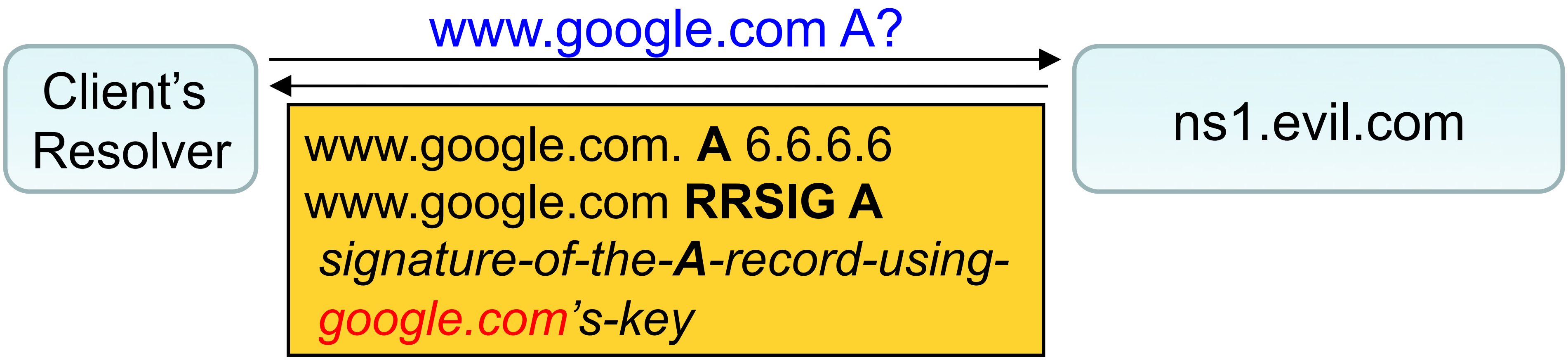
ns1.evil.com

www.google.com. **A** 6.6.6.6

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

(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!



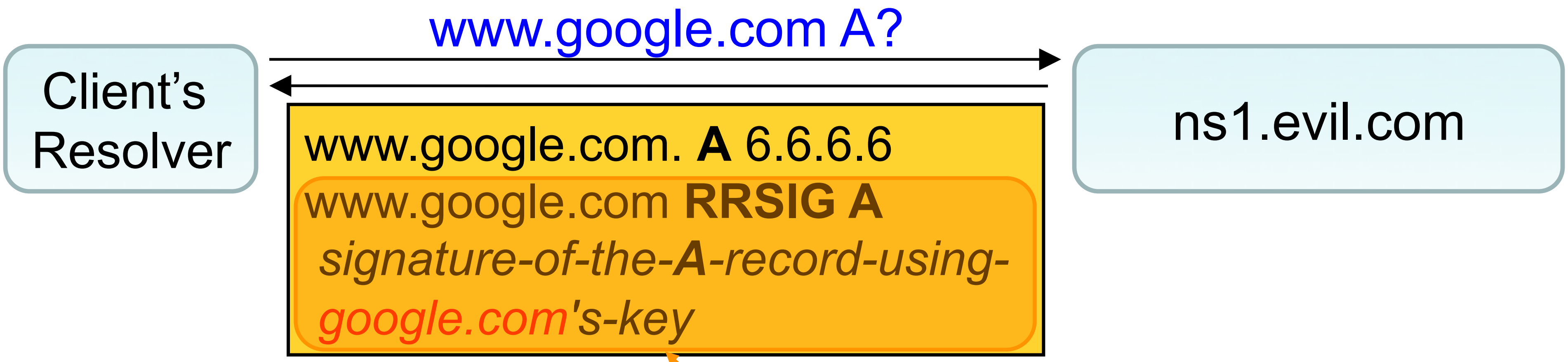
Client's Resolver

www.google.com A?

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*

DNSSEC - Mallory attacks!



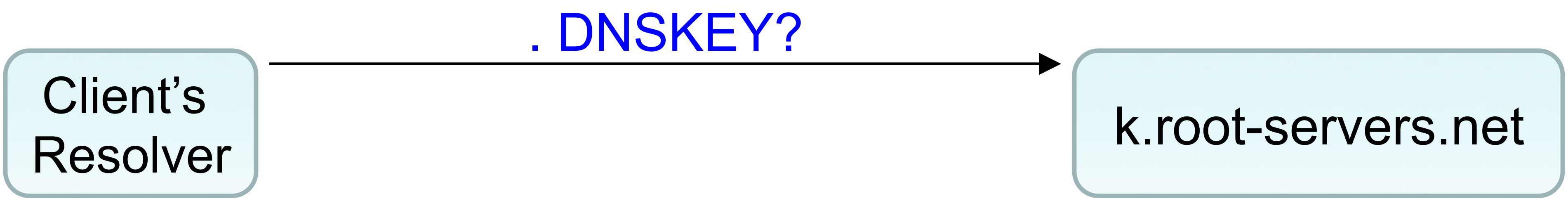
If signature **actually** comes from `google.com`'s key, resolver will believe it …

… 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

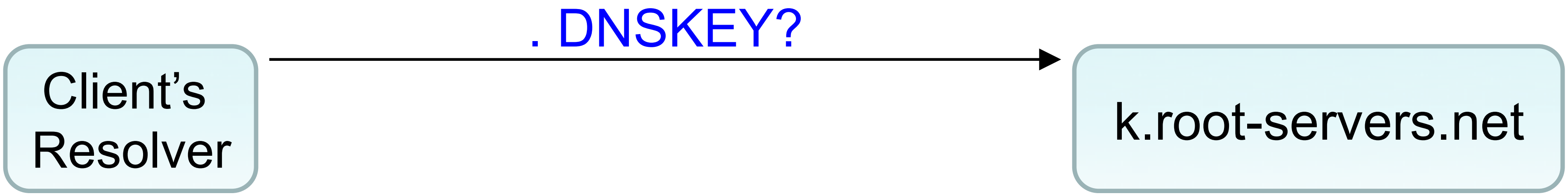
DNSSEC: Accessing keys



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



. DNSKEY?

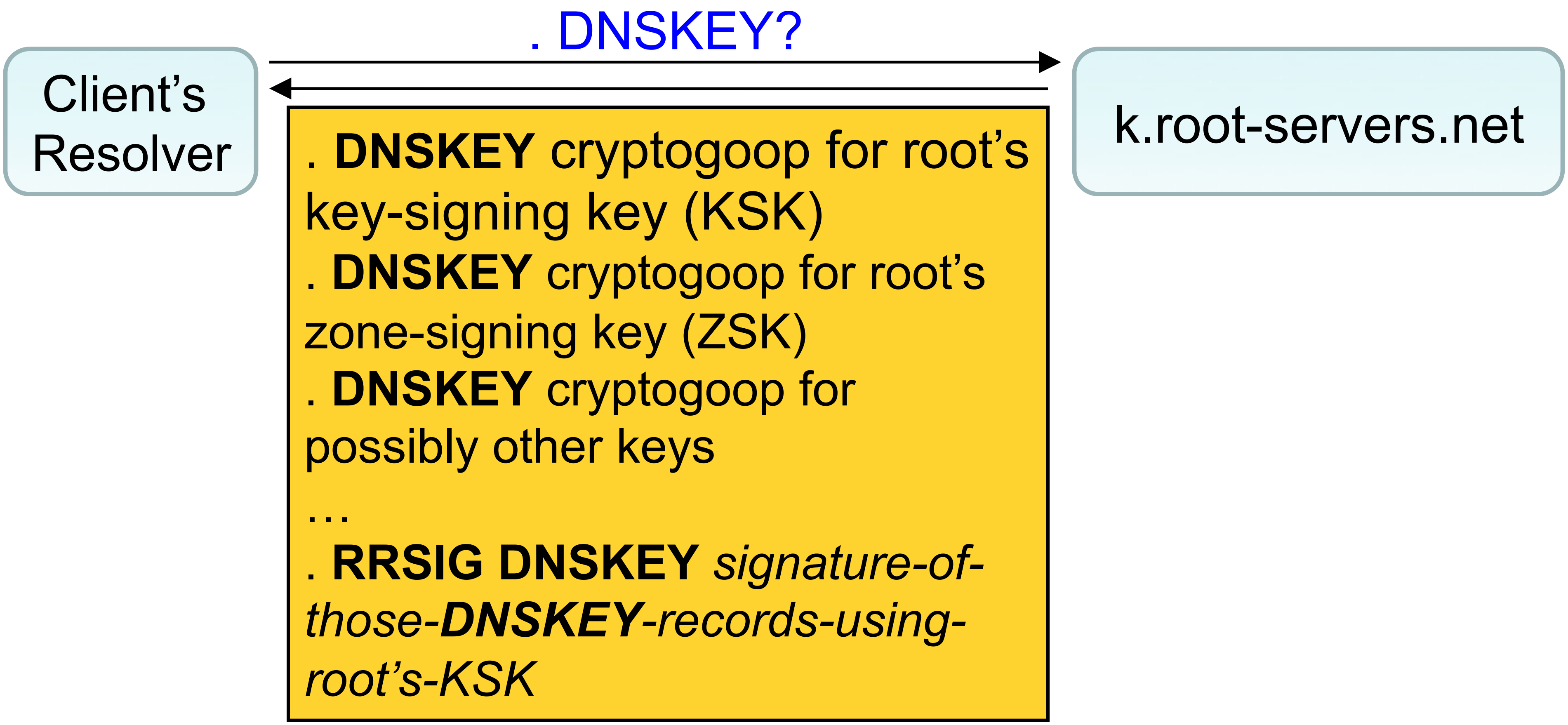
Client's Resolver

k.root-servers.net

We can ask for any other keys we need, such as `.com`'s and `google.com`'s, in parallel.

Very quickly we'll have most of the keys we need in our cache.

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*

DNSSEC: Accessing keys

Client's Resolver

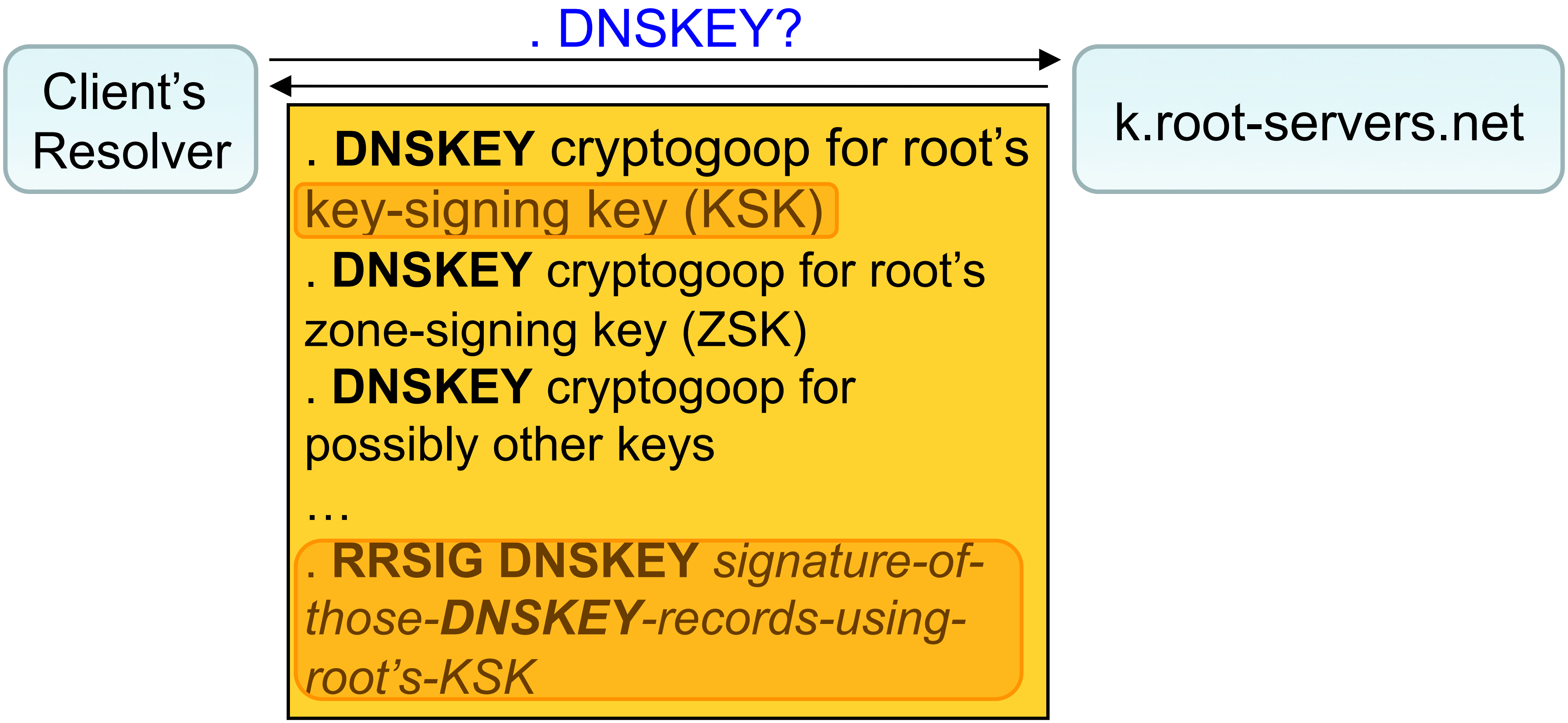
. DNSKEY?

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*

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

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)

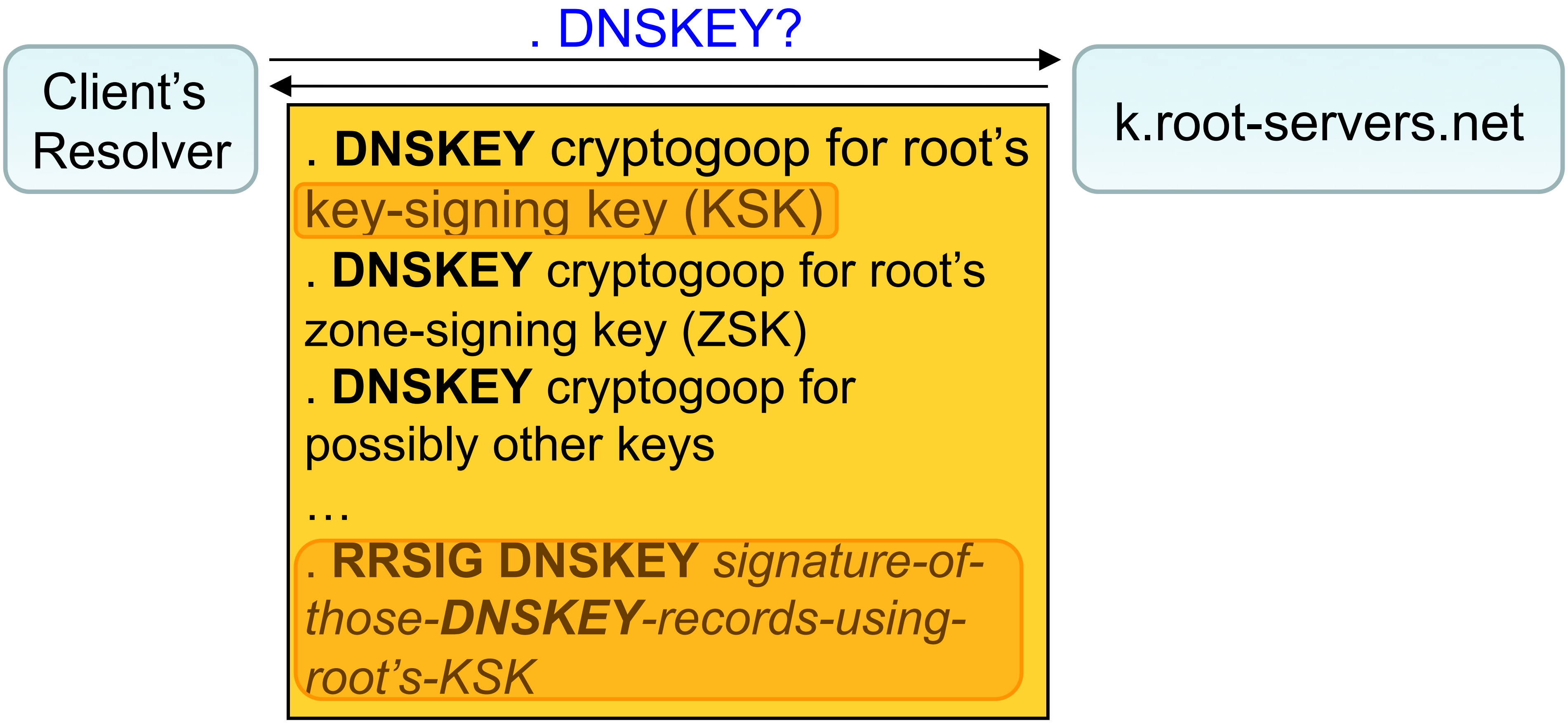
. **DNSKEY** cryptogoop for possibly other keys

…

. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

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

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)

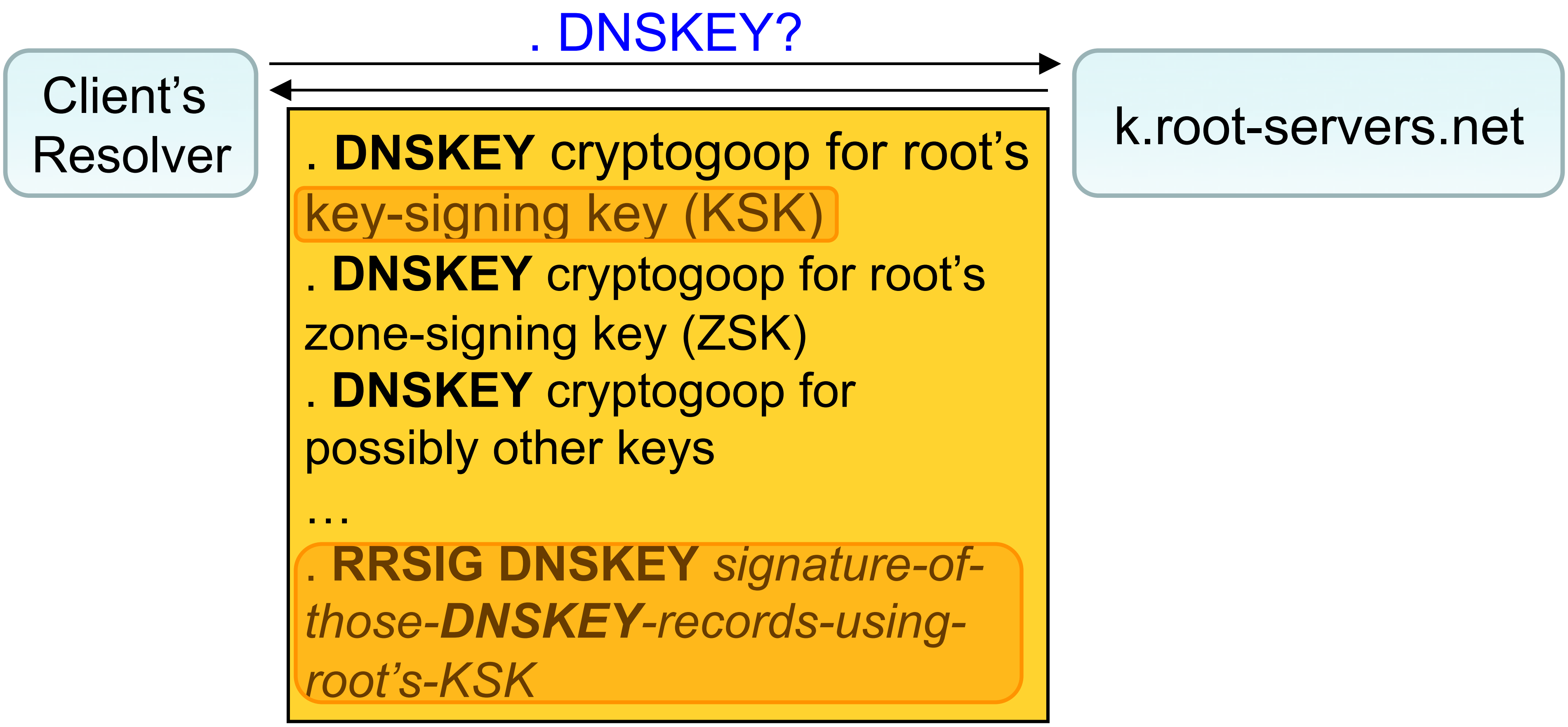
. **DNSKEY** cryptogoop for possibly other keys

…

. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

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

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)

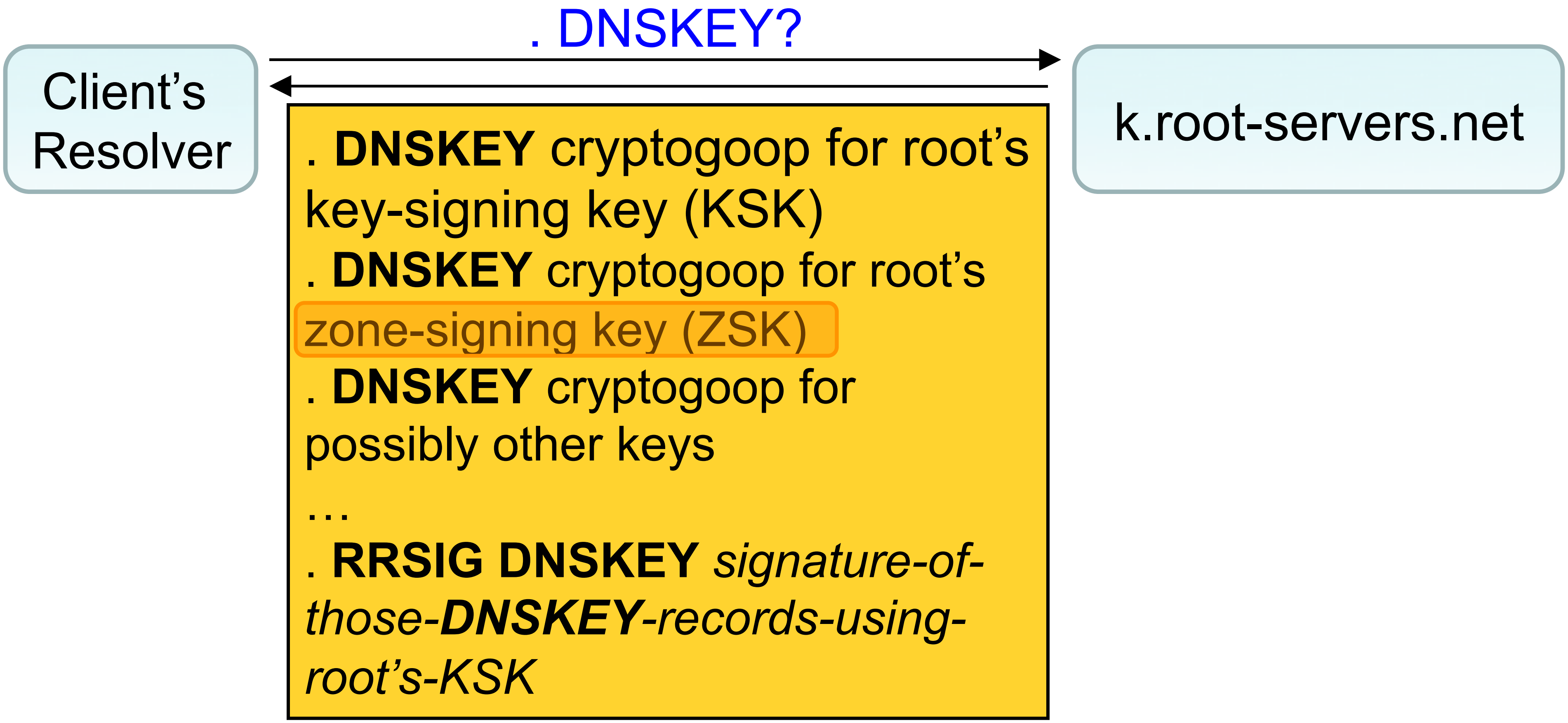
. **DNSKEY** cryptogoop for possibly other keys

…

. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

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



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)

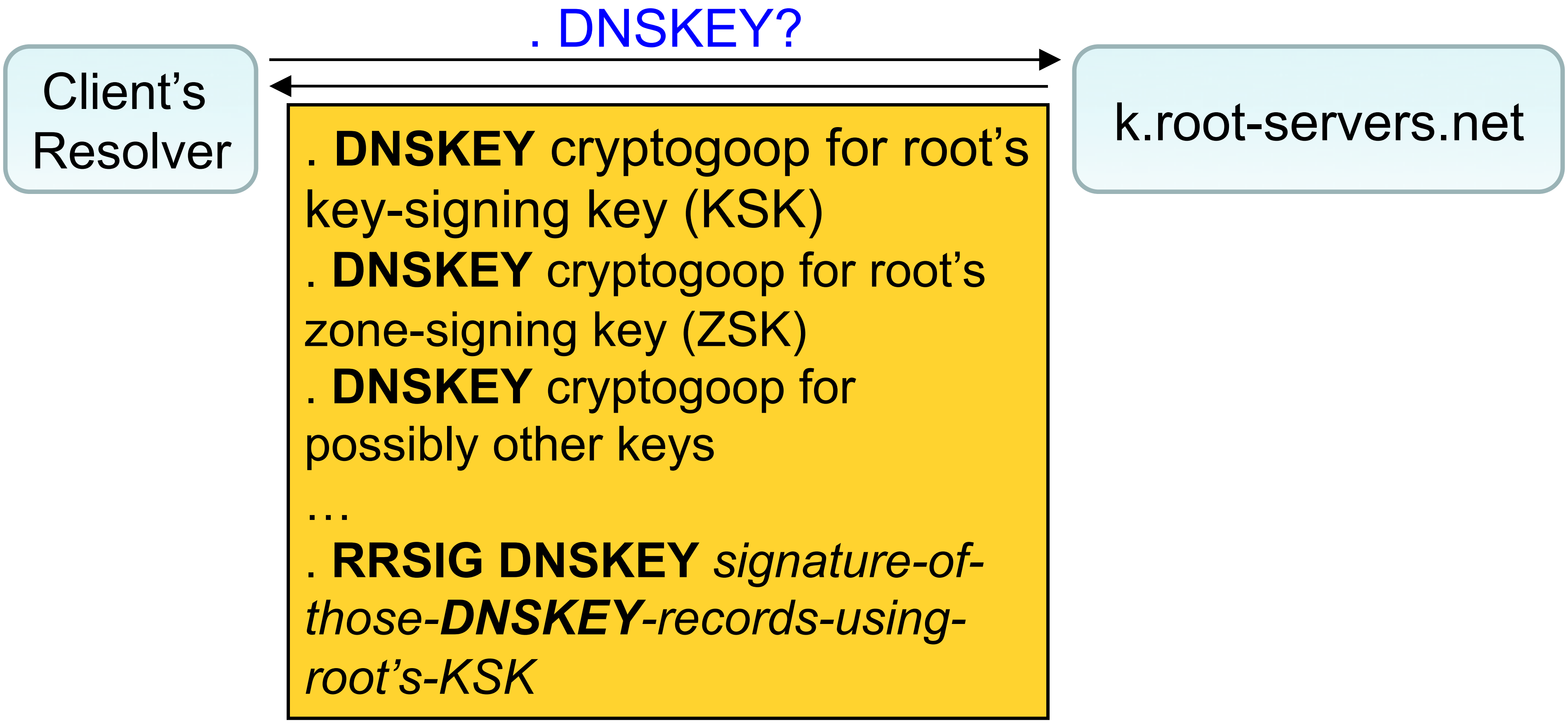
. **DNSKEY** cryptogoop for possibly other keys

…

. **RRSIG DNSKEY** *signature-of-those-**DNSKEY**-records-using-root's-KSK*

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



Client's Resolver

. DNSKEY?

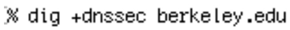
k.root-servers.net

. **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*

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*.

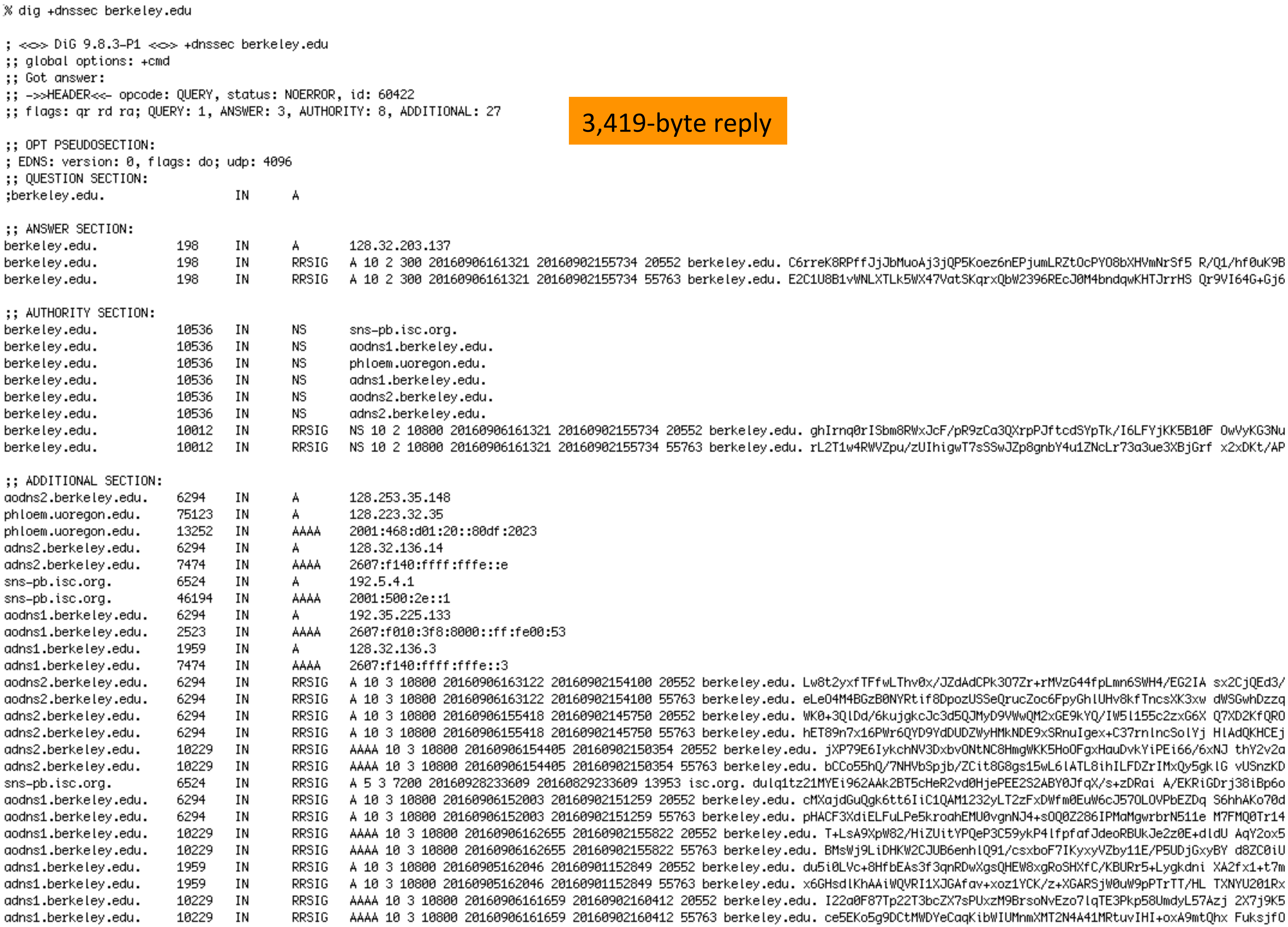
Issues With DNSSEC ?

- Issue #1: Replies are Big



```
% dig +dnssec berkeley.edu
```

69-byte query: “`dig +dnssec berkeley.edu`”



```
% dig +dnssec berkeley.edu

; <<>> DiG 9.8.3-P1 <<>> +dnssec berkeley.edu
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60422
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 8, ADDITIONAL: 27

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;berkeley.edu.                  IN      A

;; ANSWER SECTION:
berkeley.edu.           198     IN      A       128.32.203.137
berkeley.edu.           198     IN      RRSIG   A 10 2 300 20160906161321 20160902155734 20552 berkeley.edu. C6rreK8RPffJjJbMuoAj3jQP5Koez6nEPjumLRZtOcPYO8bXHVmNrSf5 R/Q1/hf0uK9B
berkeley.edu.           198     IN      RRSIG   A 10 2 300 20160906161321 20160902155734 55763 berkeley.edu. E2C1U8B1vWNLXTLk5WX47VatSKqrxQbW2396REcJ0M4bndqwKHTJrrHS Qr9VI64G+Gj6

;; AUTHORITY SECTION:
berkeley.edu.           10536   IN      NS      sns-pb.isc.org.
berkeley.edu.           10536   IN      NS      aodns1.berkeley.edu.
berkeley.edu.           10536   IN      NS      phloem.uoregon.edu.
berkeley.edu.           10536   IN      NS      adns1.berkeley.edu.
berkeley.edu.           10536   IN      NS      aodns2.berkeley.edu.
berkeley.edu.           10536   IN      NS      adns2.berkeley.edu.
berkeley.edu.           10012   IN      RRSIG   NS 10 2 10800 20160906161321 20160902155734 20552 berkeley.edu. ghIrnq0rISbm8RWxJcF/pR9zCa3QXrpPJftcdSYpTk/I6LFYjKK5B10F OwVyKG3Nu
berkeley.edu.           10012   IN      RRSIG   NS 10 2 10800 20160906161321 20160902155734 55763 berkeley.edu. rL2T1w4RWVZpu/zUIhigwT7sSSwJZp8gnbY4u1ZNcLr73a3ue3XBjGrf x2xDKt/AP

;; ADDITIONAL SECTION:
aodns2.berkeley.edu.    6294    IN      A       128.253.35.148
phloem.uoregon.edu.     75123   IN      A       128.223.32.35
phloem.uoregon.edu.     13252   IN      AAAA    2001:468:d01:20::80df:2023
adns2.berkeley.edu.     6294    IN      A       128.32.136.14
adns2.berkeley.edu.     7474    IN      AAAA    2607:f140:ffff:fffe::e
sns-pb.isc.org.         6524    IN      A       192.5.4.1
sns-pb.isc.org.         46194   IN      AAAA    2001:500:2e::1
aodns1.berkeley.edu.    6294    IN      A       192.35.225.133
aodns1.berkeley.edu.    2523    IN      AAAA    2607:f010:3f8:8000::ff:fe00:53
adns1.berkeley.edu.     1959    IN      A       128.32.136.3
adns1.berkeley.edu.     7474    IN      AAAA    2607:f140:ffff:fffe::3
aodns2.berkeley.edu.    6294    IN      RRSIG   A 10 3 10800 20160906163122 20160902154100 20552 berkeley.edu. Lw8t2yxfTFfwLThv0x/JZdAdCPk3O7Zr+rMVzG44fpLmn6SWH4/EG2IA sx2CjQEd3/
aodns2.berkeley.edu.    6294    IN      RRSIG   A 10 3 10800 20160906163122 20160902154100 55763 berkeley.edu. eLeO4M4BGzB0NYRtif8DpozUSSeQrucZoc6FpyGhlUHv8kfTncsXK3xw dWSGwhDzzq
adns2.berkeley.edu.     6294    IN      RRSIG   A 10 3 10800 20160906155418 20160902145750 20552 berkeley.edu. WK0+3QlDd/6kujgkcJc3d5QJMyD9VWwQM2xGE9kYQ/IW5l155c2zxG6X Q7XD2KfQRO
adns2.berkeley.edu.     6294    IN      RRSIG   A 10 3 10800 20160906155418 20160902145750 55763 berkeley.edu. hET89n7x16PWr6QYD9YdDUDZWyHMkNDE9xSRnuIgex+C37rnlncSolYj HlAdQKHCEj
adns2.berkeley.edu.     10229   IN      RRSIG   AAAA 10 3 10800 20160906154405 20160902150354 20552 berkeley.edu. jXP79E6IykchNV3DxbvONtNC8HmgWKK5HoOFgxHauDvkYiPEi66/6xNJ thY2v2a
adns2.berkeley.edu.     10229   IN      RRSIG   AAAA 10 3 10800 20160906154405 20160902150354 55763 berkeley.edu. bCCo55hQ/7NHVbSpjb/ZCit8G8gs15wL6lATL8ihILFDZrIMxQy5gklG vUSnzKD
sns-pb.isc.org.         6524    IN      RRSIG   A 5 3 7200 20160928233609 20160829233609 13953 isc.org. dulq1tz21MYEi962AAk2BT5cHeR2vd0HjePEE2S2ABY0JfqX/s+zDRai A/EKRiGDrj38iBp6o
aodns1.berkeley.edu.    6294    IN      RRSIG   A 10 3 10800 20160906152003 20160902151259 20552 berkeley.edu. cMXajdGuQgk6tt6IiC1QAM1232yLT2zFxDWfm0EuW6cJ57OLOVPbEZDq S6hhAKo70d
aodns1.berkeley.edu.    6294    IN      RRSIG   A 10 3 10800 20160906152003 20160902151259 55763 berkeley.edu. pHACF3XdiELFuLPe5kroahEMU0vgnNJ4+sOQ0Z286IPMaMgwrbrN511e M7FMQ0Tr14
aodns1.berkeley.edu.    10229   IN      RRSIG   AAAA 10 3 10800 20160906162655 20160902155822 20552 berkeley.edu. T+LsA9XpW82/HiZUitYPQeP3C59ykP4lfpfafJdeoRBUkJe2z0E+dldU AqY2ox5
aodns1.berkeley.edu.    10229   IN      RRSIG   AAAA 10 3 10800 20160906162655 20160902155822 55763 berkeley.edu. BMsWj9LiDHKW2CJUB6enhlQ91/csxboF7IKyxyVZby11E/P5UDjGxyBY d8ZC0iU
adns1.berkeley.edu.     1959    IN      RRSIG   A 10 3 10800 20160905162046 20160901152849 20552 berkeley.edu. du5i0LVc+8HfbEAs3f3qnRDwXgsQHEW8xgRoSHXfC/KBURr5+Lygkdni XA2fx1+t7m
adns1.berkeley.edu.     1959    IN      RRSIG   A 10 3 10800 20160905162046 20160901152849 55763 berkeley.edu. x6GHsdlKhAAiWQVRI1XJGAfav+xoz1YCK/z+XGARSjW0uW9pPTrTT/HL TXNYU201Rx
adns1.berkeley.edu.     10229   IN      RRSIG   AAAA 10 3 10800 20160906161659 20160902160412 20552 berkeley.edu. I22a0F87Tp22T3bcZX7sPUxzM9BrsoNvEzo7lqTE3Pkp58UmdyL57Azj 2X7j9K5
adns1.berkeley.edu.     10229   IN      RRSIG   AAAA 10 3 10800 20160906161659 20160902160412 55763 berkeley.edu. ce5EKo5g9DCtMWDYeCaqKibWIUMnmXMT2N4A41MRtuvIHI+oxA9mtQhx FuksjfO
```

3,419-byte reply

Issues With DNSSEC ?

- Issue #1: Replies are Big
  - E.g., “`dig +dnssec berkeley.edu`” can return 3400+ B
  - DoS amplification
  - Increased latency on low-capacity links
  - Headaches w/ older libraries that assume replies < 512B

Issues With DNSSEC ?

- Issue #1: Replies are Big
  - E.g., “`dig +dnssec berkeley.edu`” can return 3400+ B
  - 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 :-(