DNS-Based User Tracking

Amit Klein

Joint research with Benny Pinkas

Center for Research in Applied
Cryptography and Cyber Security

Last update: 2019-02-23 18:22 UTC+0200

Why do we need user tracking?

From the literature:

- **Real-time targeted marketing**
- **Campaign measurement**
- **Fraud detection**
- **Protection against account hijacking**
- Anti-bot and anti-scraping services
- Enterprise security management
- Protection against DDOS attacks
- Reaching customers across devices
- Limiting number of accesses to services

User tracking in 1999…

- Cookies!
- Later, also: localStorage and friends
- Two browsers (IE+Mozilla), one OS (Windows)

User tracking in 2019 – the challenges

- **Privacy mode** boundary
- Identical HW+SW (the "**golden image**" problem)
- **Many browsers** (IE, FF, GC, Safari) on desktops/laptops (Windows, macOS) and mobile (iOS, Android)
- **Awareness** – history clearing, browser restart, browser per task

Have the cake and eat it too

	Fingerprinting (typical)	Tagging (typical)
Privacy mode boundary	✓	✗
Identical HW+SW	✗	✓
Coverage	?	?
History cleanup	✓	✗
Browser restart	✓	✓
Cross-browser	✓	✗

Have the cake and eat it too

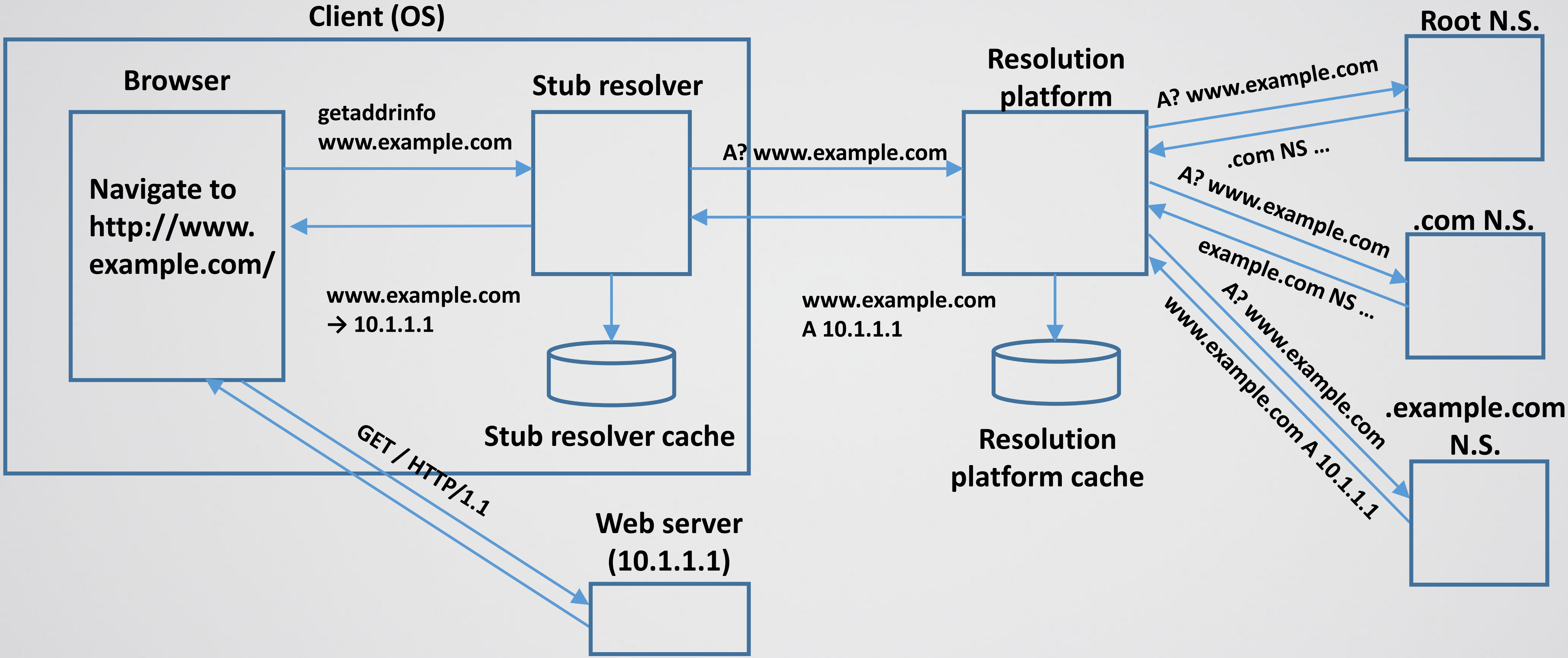
	Fingerprinting (typical)	Tagging
Privacy mode boundary	✓	
Identical HW+SW	✗	✓
Coverage	?	?
History cleanup		✗
Browser restart		✓
Cross-browser	✓	✗

We designed a scheme that satisfies all six requirements

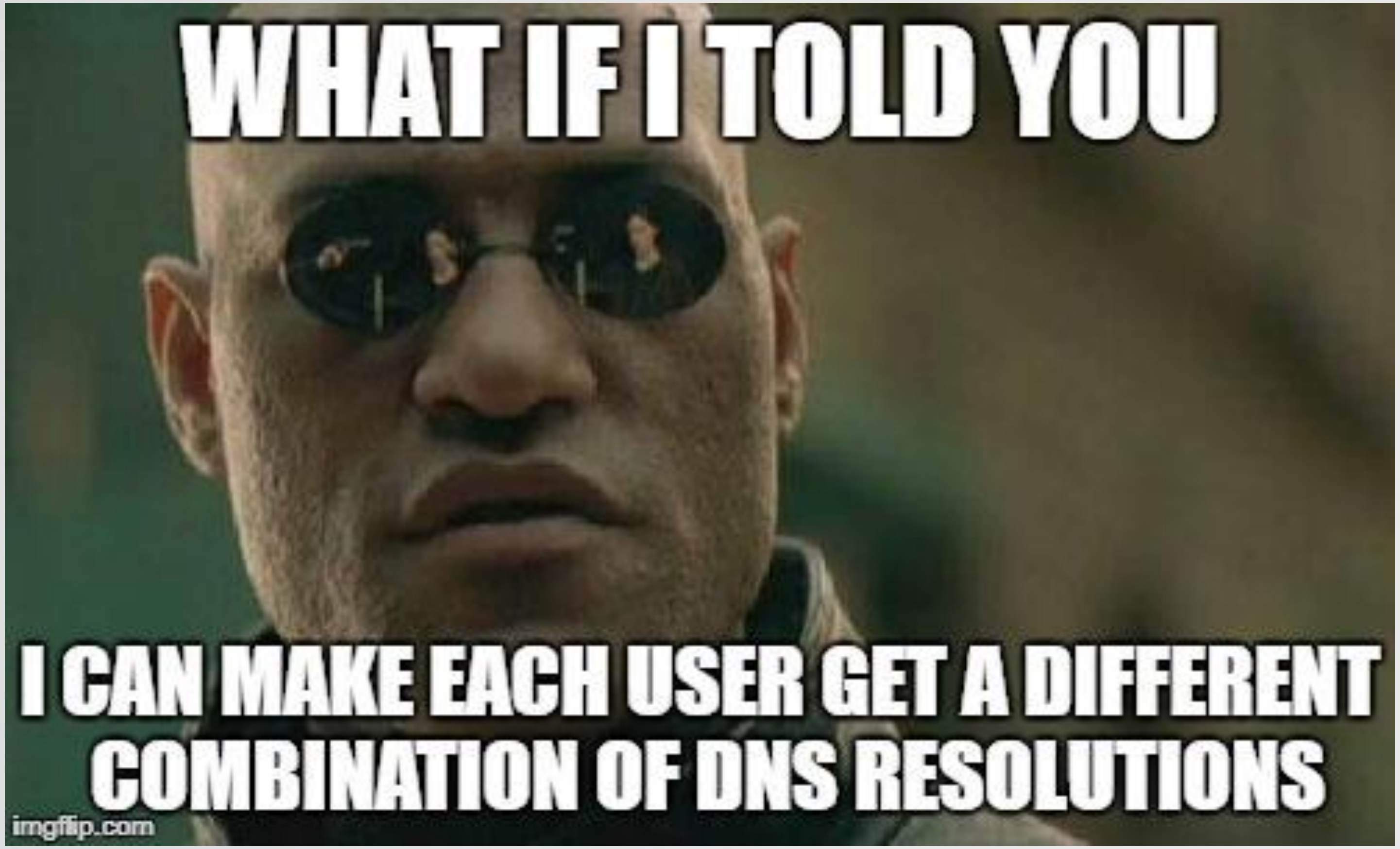
Have the cake and eat it too

- We devised a technique that basically satisfies all 6 requirements
- DNS-based (duh)
- Some disclaimers:
  - Good coverage (resolver SW), but not perfect
  - Cross browser works, but not in some browser combinations
  - Doesn’t work across network switches (and OS restart)
  - TTL limitations

DNS refresher



DNS-based user tracking



The main idea (example)

- User 1:
  - x_1.anonymity.fail → 10.4.5.6, … (2)
  - x_2.anonymity.fail → 10.1.2.3, … (1)
  - x_3.anonymity.fail → 10.7.8.9, … (3)
  - …

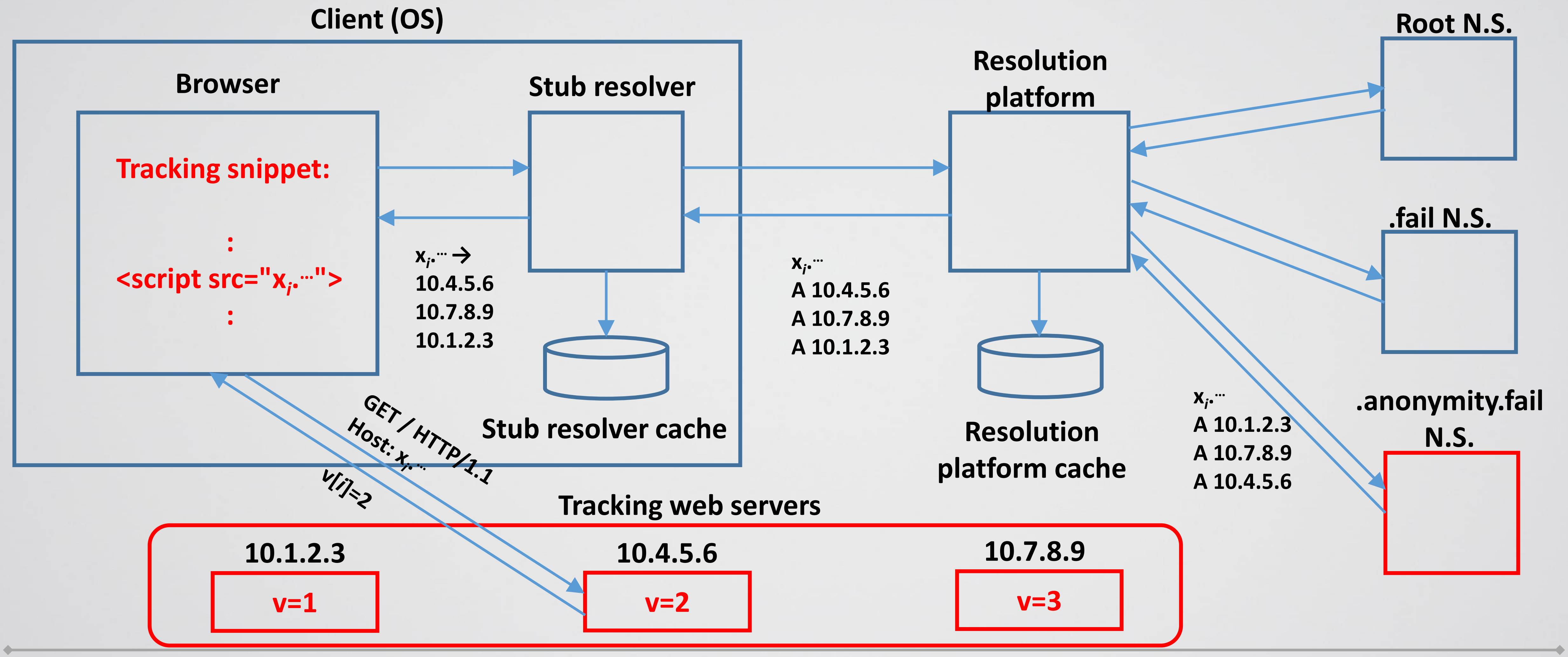
  ID=(2,1,3,…)
- User 2:
  - x_1.anonymity.fail → 10.1.2.3, … (1)
  - x_2.anonymity.fail → 10.1.2.3, … (1)
  - x_3.anonymity.fail → 10.4.5.6, … (2)
  - …

  ID=(1,1,2,…)

The main idea - components

- Tracking is carried out via an HTML+Javascript "**snippet**" which you can place in any page.
- The snippet references Javascript resources on multiple hosts (x_1,…,x_N) in the **tracking domain** (managed via a **dedicated auth. name server**).
- The tracker also runs a **web server farm**. Each web server *j* has a dedicated IP address and returns a different JS code (e.g. `v[i]=`*j*)

The main idea - technique



Mandatory requirements

- **Req #1**: Same client must get same ID each time (for a reasonable time)
  - Caching at the Stub Resolver ensures this
- **Req #2**: Different clients must get different IDs
  - This is obvious for clients that use different DNS resolvers (each resolver gets its own order of IPs)
  - But what happens with clients behind the same resolver?

IDs in the same farm

- Main problem: the answer (list of IP addresses) is cached in the resolver itself!
- So theoretically, the resolver returns the same response to all its clients (and they all get the same ID). Right?
  - Not necessarily. BIND 9.x (the most popular SW) randomizes the order!
  - Microsoft DNS server, MaraDNS do round robin – we can still use this.
  - Unbound, PowerDNS – fixed order (bad). But a very small portion of the landscape.

IDs in the same farm – multiple resolvers

- Load-balanced “farm” of resolvers works in the tracker’s favor!
- Clients are load balanced over resolvers, so even if a single resolver does return data in the same order, load balancing among resolvers provides the necessary randomness

Complications and limitations

- **Windows: dual cache**: IE/Edge+Firefox, vs. Chrome+Opera
- **macOS: Chrome has its own stub resolver** (but Safari and Firefox share the stub resolver cache)
- **TTL cap** – most resolvers put a cap on the TTL (7d-¼d), stub resolvers as well.
- **Disconnecting** from the network automatically flushes the stub resolver DNS cache
- **Restarting** the machine flushes the DNS cache

How do we score?

- Privacy mode boundary – **GOOD**. Both modes use the stub resolver cache.
- Identical HW+SW – **GOOD**. Each device gets a random ID.
- Coverage – **PRETTY GOOD**. Except for single Unboud resolver or single PowerDNS>3.6 resolver. Coverage >90% for enterprises.
- History cleanup – **GOOD**. Doesn’t touch the stub resolver cache (except Chrome on macOS).
- Browser restart – **GOOD**. Ditto.
- Cross browser – **GOOD**. Except Chrome on macOS, and the dual cache on Windows.

Mitigations

- Systematic solution (need both):
  - Browsers use random IP from RRset for each new connection
    - Takes care of the "randomized" RRset approach (|RRset|>1)
  - Sticky-by-client (IP) DNS load balancing
    - Takes care of the load-balancing approach with |RRset|=1 (there'll be only |resolvers| possible IDs)
- Forward shared HTTP proxy (or Tor)
- Flush DNS cache very often
- Tracking domain blacklisting (cat and mouse)

Conclusions

- A new user tracking method:
  - DNS Based
  - Crosses the privacy mode boundary
  - Handles the golden image challenge
  - Has good coverage
- Not easy to mitigate!
- Additional results (non-DNS-tracking):
  - DNS load balancing strategies (good for connecting to a specific resolver)
  - Systematic info about resolver SW, stub resolver SW, browser DNS behavior

Q&A

Thanks!