Loophole

Timing Attacks on Shared Event Loops in Chrome

Pepe Vila

November 22, 2016

Introduction

- Event-driven programming
- Event loops
- A timing side-channel on event loops

Introduction: Event-driven programming

EDP is a **programming paradigm** for GUI, web clients, networks and server-side

[1] https://html.spec.whatwg.org/#event-loop

Introduction: Event-driven programming

EDP is a **programming paradigm** for GUI, web clients, networks and server-side

The flow of the program is determined by **events** or **messages**

[1] https://html.spec.whatwg.org/#event-loop

Introduction: Event-driven programming

EDP is a **programming paradigm** for GUI, web clients, networks and server-side

The flow of the program is determined by **events** or **messages**

Examples:

- Nginx, Node.js or memcached
- Used for message passing: inter-(thread | process) communication
- HTML5 standard [1] mandates user agents to use EDP:

[1] https://html.spec.whatwg.org/#event-loop

Introduction: Event loops

Event loop, message dispatcher, message loop, or run loop

Introduction: Event loops

Event loop, message dispatcher, message loop, or run loop

- FIFO queue & dispatcher:

```
Q = [];
while (true) {
    M = Q.shift(); // dequeue
    process(M);
}
```

Introduction: Event loops

Event loop, message dispatcher, message loop, or run loop

- FIFO queue & dispatcher:

```
Q = [];
while (true) {
    M = Q.shift(); // dequeue
    process(M);
}
```

- If **queue** is empty, waits until an event arrives

Introduction: Event loops

Event loop, message dispatcher, message loop, or run loop

- FIFO queue & dispatcher:

```
Q = [];
while (true) {
    M = Q.shift(); // dequeue
    process(M);
}
```

- If **queue** is empty, waits until an event arrives
- Blocking operations (e.g., database and network requests) are dealt with **asynchronously**

Introduction: Event loops

Event loop, message dispatcher, message loop, or run loop

- FIFO queue & dispatcher:

```
Q = [];
while (true) {
    M = Q.shift(); // dequeue
    process(M);
}
```

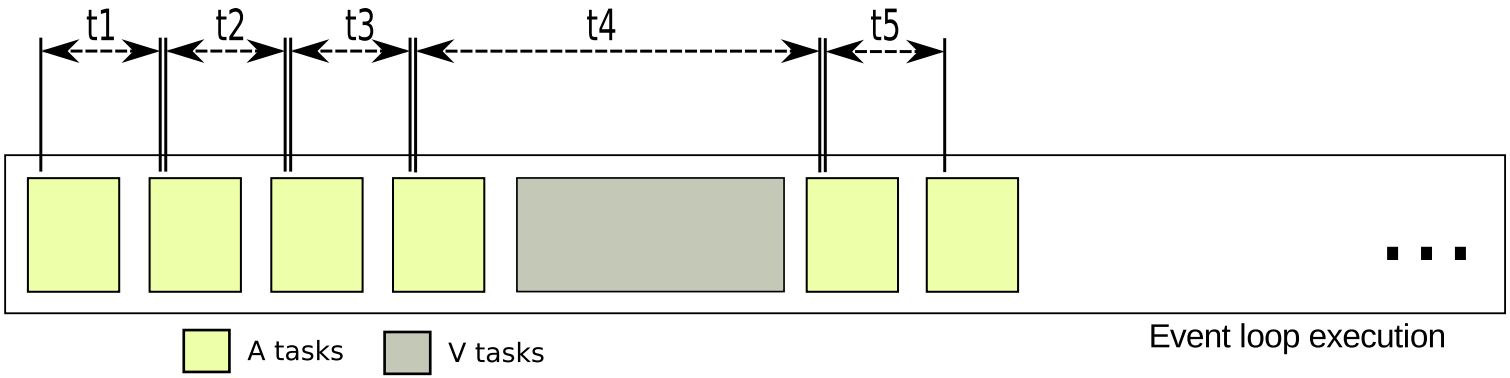
- If **queue** is empty, waits until an event arrives
- Blocking operations (e.g., database and network requests) are dealt with **asynchronously**
- Simple **concurrency model** for programmers

Introduction: A timing side-channel on event loops

Event loops are susceptible to **timing side-channel** attacks:

Introduction: A timing side-channel on event loops

Event loops are susceptible to **timing side-channel** attacks:



when **shared** between mutually distrusting programs

Our work (in poetry)

“Loophole”

Exploit a timing side-channel

in the Chrome web browser

to break user privacy

using machine learning techniques

- Abraham Lincoln

Chrome's architecture

- Same Origin Policy (SOP)
- Multi-process
- Shared event loops

Chrome's architecture: Same Origin Policy (SOP)

- Central concept in the web security model
- Script from a site A can not access data from site V if **origins** differ:

$$\text{Origin} := (\text{scheme, domain, port})$$

Chrome's architecture: Same Origin Policy (SOP)

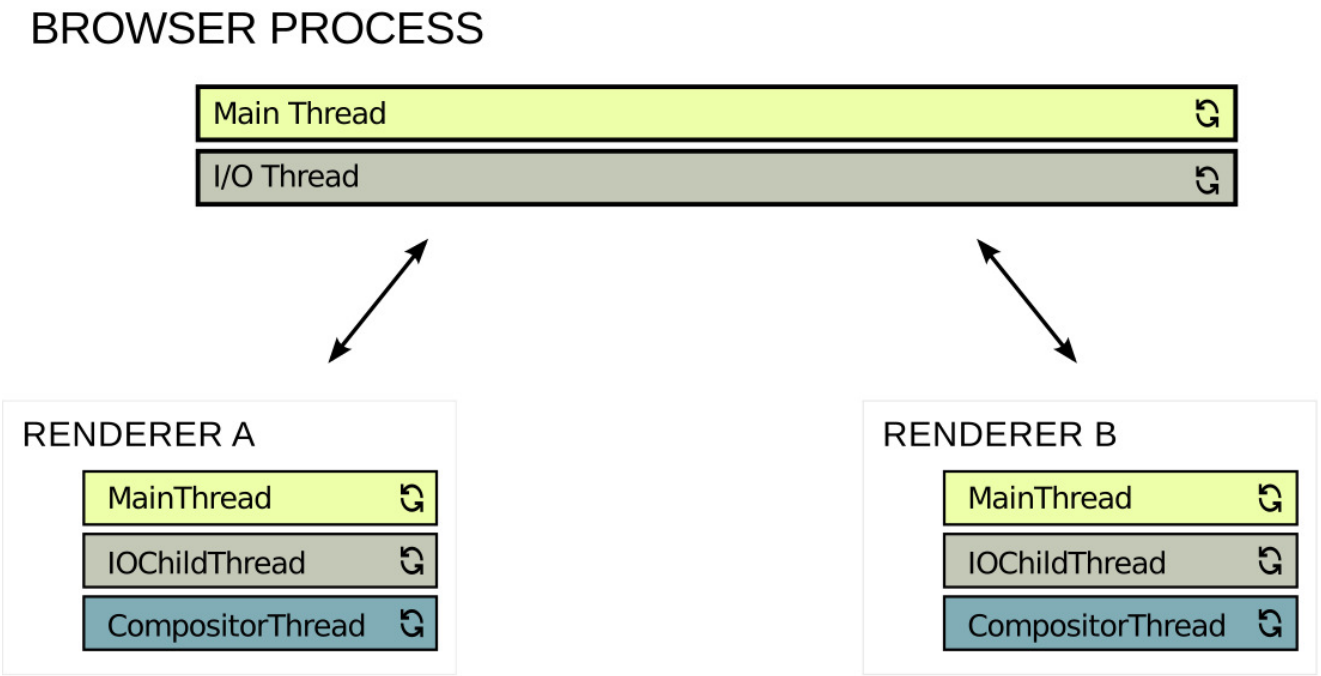
- Central concept in the web security model
- Script from a site A can not access data from site V if **origins** differ:

$$\text{Origin} := (\text{scheme, domain, port})$$

Origin 1	*Origin 2*
http://example.com:8080	http://example.com
http://mail.example.com	http://app.example.com
https://foo.example.com	https://foo.example.com
https://example.com	http://example.com

Chrome's architecture: Multi-process

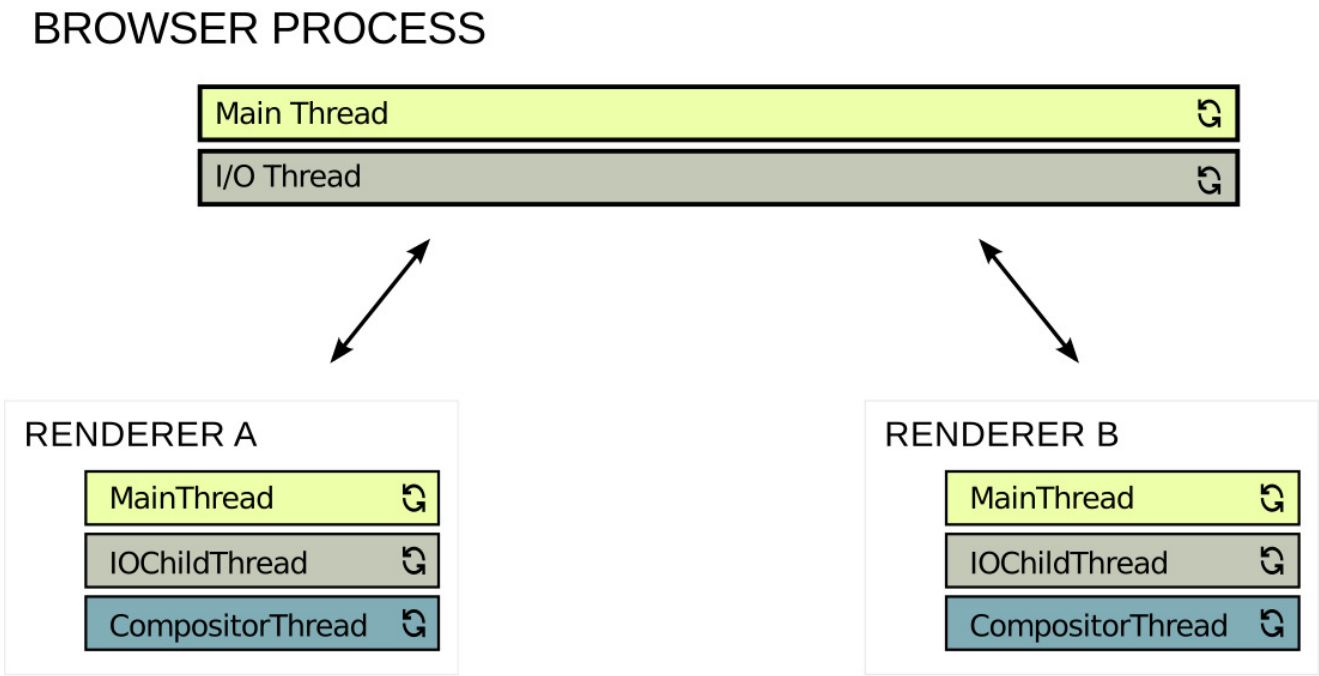
- Multi-process: 1 privileged **host** — N sandboxed **renderers**



[2] Chrome's implementation of an event loop

Chrome's architecture: Multi-process

- Multi-process: 1 privileged **host** — N sandboxed **renderers**

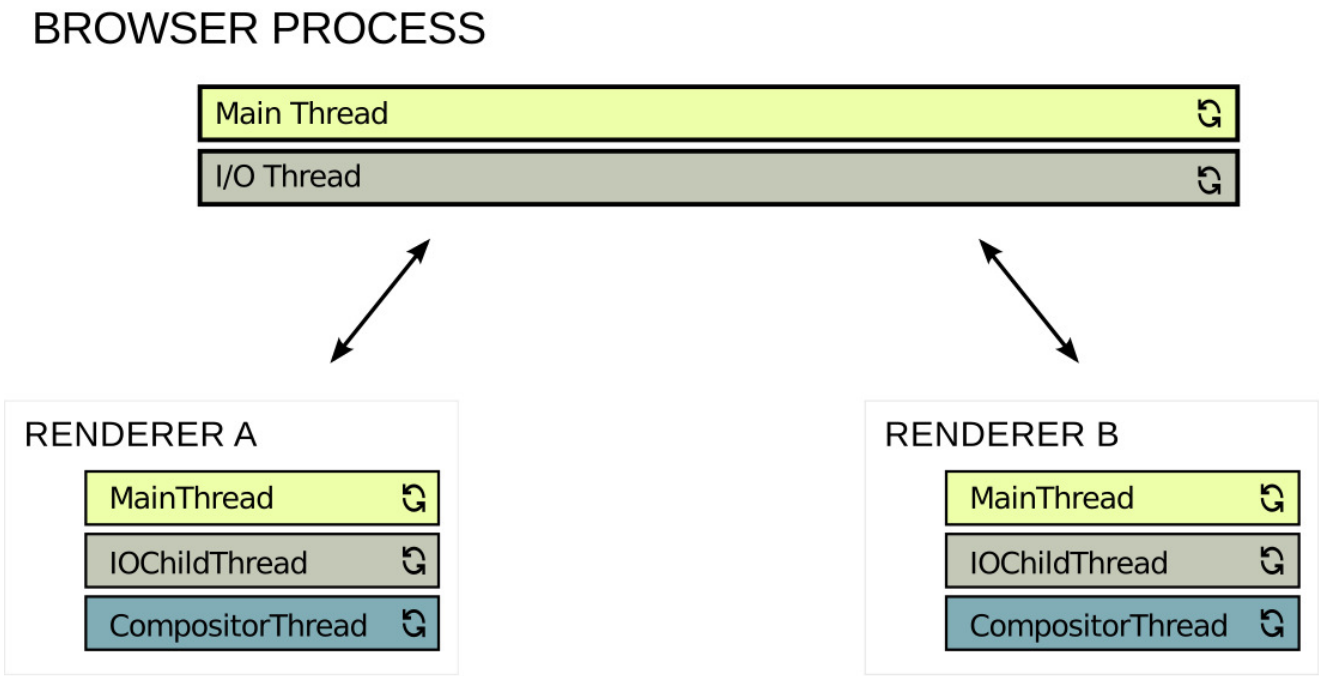


- Each process has multiple threads. Each thread one **message loop** [2]

[2] Chrome's implementation of an event loop

Chrome's architecture: Multi-process

- Multi-process: 1 privileged **host** — N sandboxed **renderers**



- Each process has multiple threads. Each thread one **message loop** [2]
- **DEMO:** Chrome's task manager

[2] Chrome's implementation of an event loop

Chrome's architecture: Shared event loops

- Different policies for mapping applications into renderer processes (default: *process-per-site-instance*)
- A **Site** is a registered domain plus a scheme

Chrome's architecture: Shared event loops

- Different policies for mapping applications into renderer processes (default: *process-per-site-instance*)
- A **Site** is a registered domain plus a scheme (different than SOP)

Chrome's architecture: Shared event loops

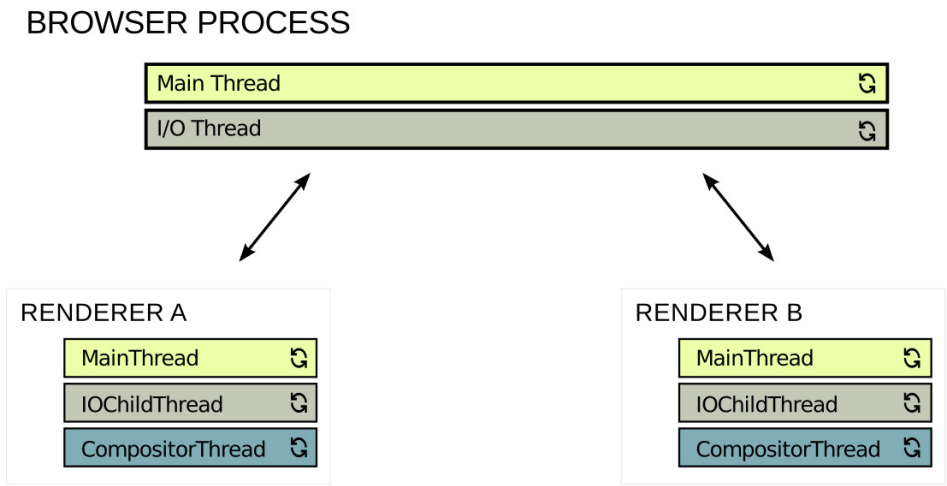
- Different policies for mapping applications into renderer processes (default: *process-per-site-instance*)
- A **Site** is a registered domain plus a scheme (different than SOP)
- Sharing the renderer
  - When using iframes, linked nagivation or $|processes| > T$
  - $T = 32$ for 4 GB of RAM, and $T = 70$ for 8 GB or more

Chrome's architecture: Shared event loops

- Different policies for mapping applications into renderer processes (default: *process-per-site-instance*)
- A **Site** is a registered domain plus a scheme (different than SOP)
- Sharing the renderer
  - When using iframes, linked nagivation or $|processes| > T$
  - $T = 32$ for 4 GB of RAM, and $T = 70$ for 8 GB or more
- Sharing the host process
  - One for all renderers
  - IPC through I/O thread

Spying on shared event loops

- Main thread of a renderer
- I/O thread of the host process



Spying on shared event loops

- **Main thread of renderer processes**

Spying on shared event loops

- **Main thread of renderer processes**
  - runs resource parsing, style calculation, layout, painting and Javascript
  - each **task** blocks the event loop for a while
  - when 2 pages share the process, the main thread's event loop is shared
  - A can eavesdrop information from V's tasks

Spying on shared event loops

- **Main thread of renderer processes**
  - runs resource parsing, style calculation, layout, painting and Javascript
  - each **task** blocks the event loop for a while
  - when 2 pages share the process, the main thread's event loop is shared
  - A can eavesdrop information from V's tasks
- **I/O thread of the host process**
  - manages IPC with all children renderers
  - demultiplexes all UI events to each corresponding renderer
  - multiplexes all network requests from renderers
  - each task/message/event also blocks the event loop

Spying on shared event loops

- **Main thread of renderer processes**
  - runs resource parsing, style calculation, layout, painting and Javascript
  - each **task** blocks the event loop for a while
  - when 2 pages share the process, the main thread's event loop is shared
  - A can eavesdrop information from V's tasks
- **I/O thread of the host process**
  - manages IPC with all children renderers
  - demultiplexes all UI events to each corresponding renderer
  - multiplexes all network requests from renderers
  - each task/message/event also blocks the event loop

Some tasks are very fast ($<< 0.1\,\text{ms}$). We need high timing resolution.

Spying on shared event loops: renderer's main thread

Monitor the event loop from an arbitrary HTML page running Javascript:

```
function loop() {
    save(performance.now()); // high-resolution timestamp
    self.postMessage(0,'*'); // recursive invocation
}
self.onmessage = loop; // set event handler
self.postMessage(0,'*'); // post first async task
```

Spying on shared event loops: renderer's main thread

Monitor the event loop from an arbitrary HTML page running Javascript:

```
function loop() {
    save(performance.now()); // high-resolution timestamp
    self.postMessage(0,'*'); // recursive invocation
}
self.onmessage = loop; // set event handler
self.postMessage(0,'*'); // post first async task
```

1. Generates a trace of timing measurements
2. Resolution $\approx$ 25 μs

Spying on shared event loops: host's I/O thread

Monitor the loop from any HTML page running Javascript:

```
function loop() {
    save(performance.now());
    fetch(new Request('http://0.0.0.0')).catch(loop);
}
loop();
```

Spying on shared event loops: host's I/O thread

Monitor the loop from any HTML page running Javascript:

```
function loop() {
    save(performance.now());
    fetch(new Request('http://0.0.0.0')).catch(loop);
}
loop();
```

Performs an invalid network request. Task is posted into the I/O event to be processed asynchronously. Fails quick and triggers our "catch" callback.

1. Resolution $\approx$ 0.5 ms

Spying on shared event loops: host's I/O thread

Monitor the loop from any HTML page running Javascript:

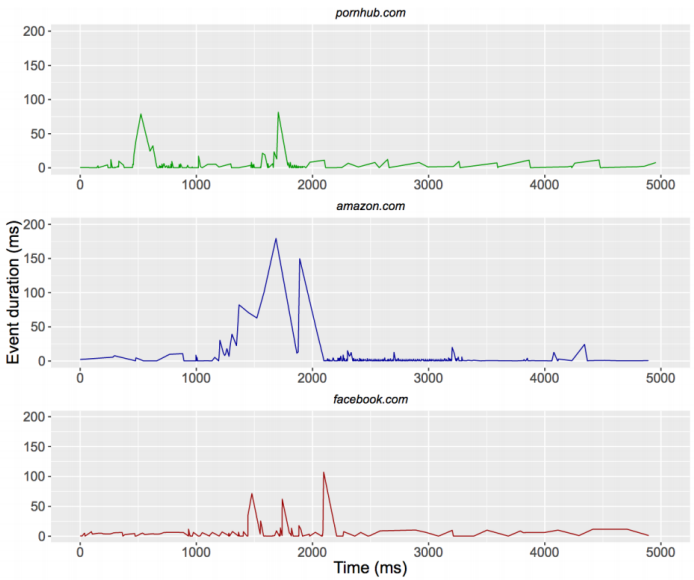
```
function loop() {
    save(performance.now());
    fetch(new Request('http://0.0.0.0')).catch(loop);
}
loop();
```

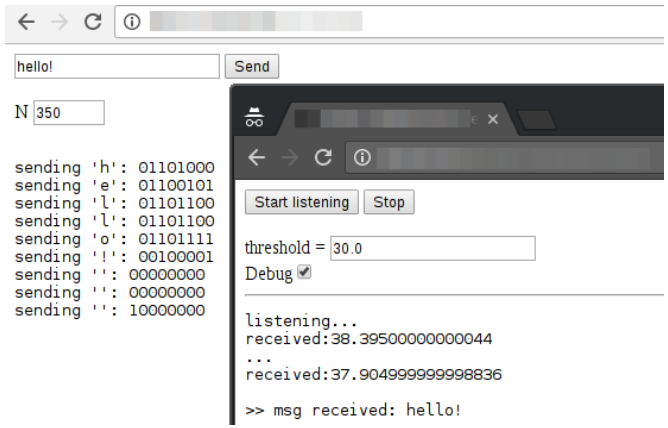
Performs an invalid network request. Task is posted into the I/O event to be processed asynchronously. Fails quick and triggers our "catch" callback.

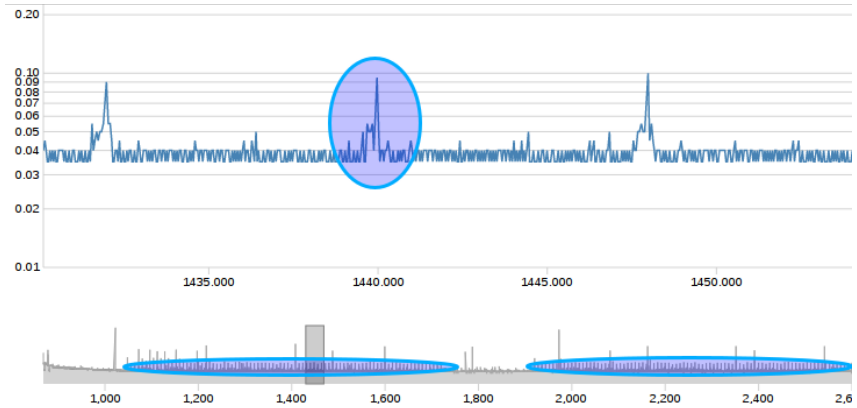
1. Resolution $\approx$ 0.5 ms
2. **NEW METHOD:** We obtain a resolution of < 0.1 ms! :D

Attacks

- Covert channel
- Web page fingerprinting
- User action detection







Attacks: covert channel

- **Covert-channel** using timing differences

  bandwidth of 200 bit/s on same renderer,

  and 5 bit/s across processes

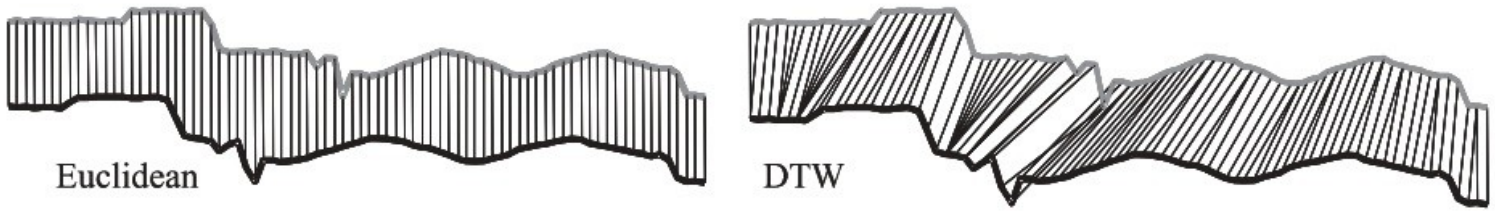
VIDEO: `https://www.youtube.com/watch?v=IlndCZmRDmI`

Attacks: web page fingerprinting

Attacks: web page fingerprinting

Dynamic Time Warping

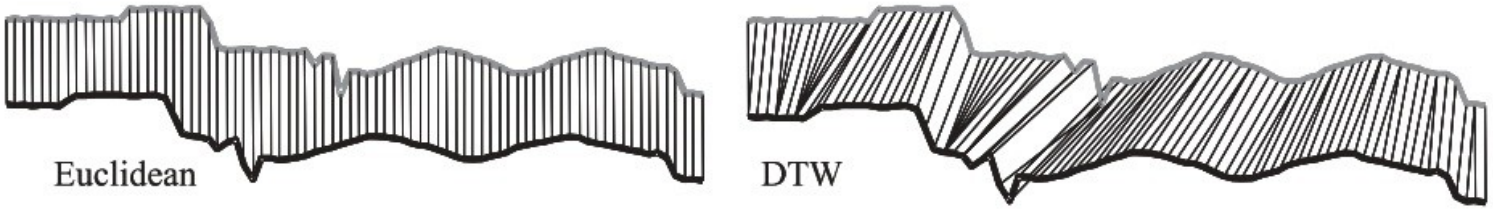
- Distance metric for **time series**: $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_m)$
- Robust to horizontal compressions and streches (warping)



Attacks: web page fingerprinting

Dynamic Time Warping

- Distance metric for **time series**: $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_m)$
- Robust to horizontal compressions and streches (warping)

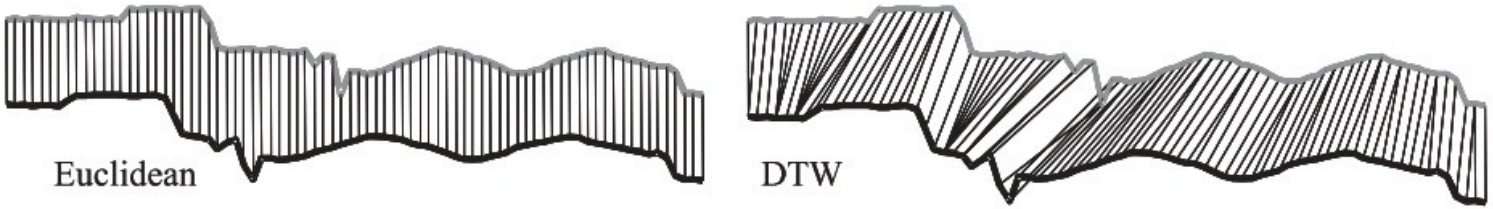


- Computes cross-distance matrix: $M(i, j) = f(x_i, y_j) \geq 0$

Attacks: web page fingerprinting

Dynamic Time Warping

- Distance metric for **time series**: $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_m)$
- Robust to horizontal compressions and streches (warping)



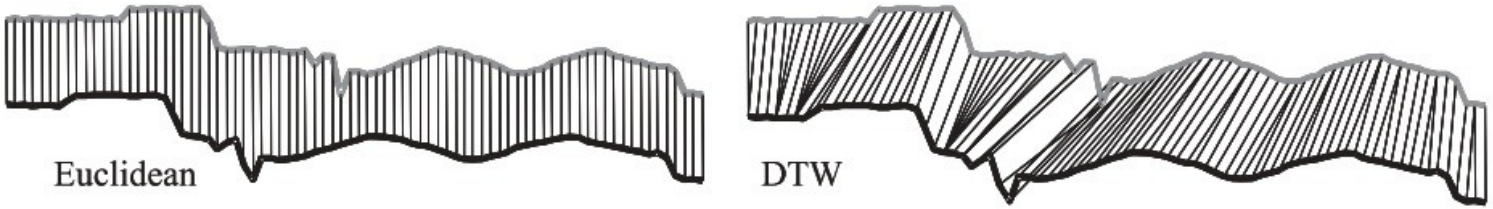
- Computes cross-distance matrix: $M(i, j) = f(x_i, y_j) \geq 0$
- Find optimal alignment ϕ such that:

$$DTW(X, Y) = \min_{\phi} d_{\phi}(X, Y)$$

Attacks: web page fingerprinting

Dynamic Time Warping

- Distance metric for **time series**: $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_m)$
- Robust to horizontal compressions and streches (warping)



- Computes cross-distance matrix: $M(i, j) = f(x_i, y_j) \geq 0$
- Find optimal alignment ϕ such that:

$$DTW(X, Y) = \min_{\phi} d_{\phi}(X, Y)$$

- Cost $\mathcal{O}(n \cdot m) \rightarrow$ We use Lemire's lower bound.

Attacks: web page fingerprinting

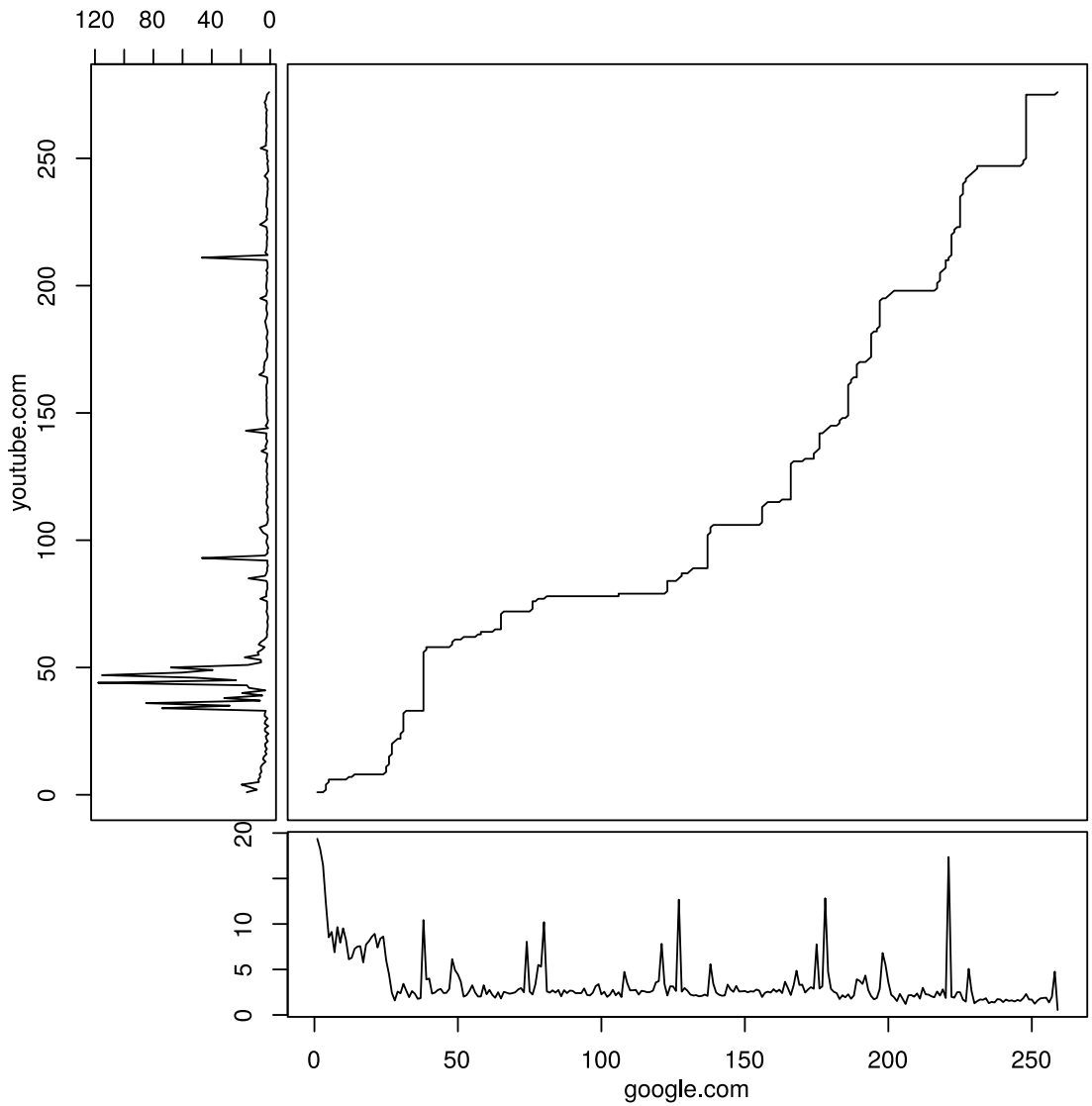


Figure: Warping matrix with optimal alignment between two time series

Attacks: web page fingerprinting

Experiments

- 500 main pages from Alexa's Top sites

Attacks: web page fingerprinting

Experiments

- 500 main pages from Alexa's Top sites
- 30 traces $\times$ page (monitoring main thread)
- 6 traces $\times$ page (monitoring IO thread)

Attacks: web page fingerprinting

Experiments

- 500 main pages from Alexa's Top sites
- 30 traces $\times$ page (monitoring main thread)
- 6 traces $\times$ page (monitoring IO thread)
- only ONE sample for training

Attacks: web page fingerprinting

Experiments

- 500 main pages from Alexa's Top sites
- 30 traces $\times$ page (monitoring main thread)
- 6 traces $\times$ page (monitoring IO thread)
- only ONE sample for training
- testing multiple configuration values

Attacks: web page fingerprinting

Experiments

- 500 main pages from Alexa's Top sites
- 30 traces $\times$ page (monitoring main thread)
- 6 traces $\times$ page (monitoring IO thread)
- only ONE sample for training
- testing multiple configuration values
- k-fold cross-validation

Attacks: web page fingerprinting

Renderer results: 65%

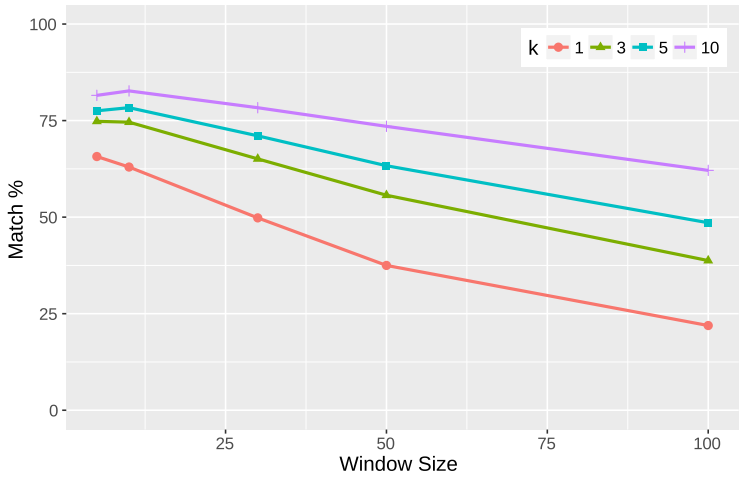


Figure: Matching rates with best configuration and multiple tolerance

Host process results: 25%

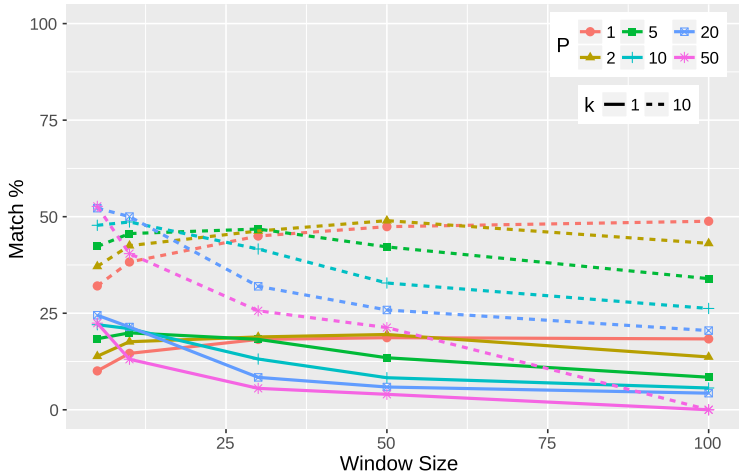


Figure: Matching rates with multiple configurations and tolerance

Attacks: user action detection

- **LoopScan** tool for visualizing event loops in real-time

  "see" mouse movement, scrolling, clicks or keystrokes in other tabs

DEMO: `http://vwzq.net/lab/ioloop/monitor.html`

Conclusions

- Resource sharing is dangerous
- It is possible to spy other tabs/pages in the same browser
- Machine learning is useful for side-channel attacks

Conclusions

- Resource sharing is dangerous
- It is possible to spy other tabs/pages in the same browser
- Machine learning is useful for side-channel attacks

Future work:

- automatize event recognition (online learning)
- pattern used by ALL modern browsers
- lots of tecnologies relying on event loops

Thank you. Questions?