Device Tracking via Linux's New TCP Source Port Selection Algorithm*

Moshe Kol, Amit Klein, Yossi Gilad
Hebrew University of Jerusalem

DANSS 2022-2023

*** Accepted to USENIX Security 2023**

Web-based User Tracking

- Web-based user (device) tracking is used for various purposes:

✗ Real-time targeted marketing

✗ Surveillance purposes

✓ Fraud detection

✓ Campaign measurement

✓ Protection against account hijacking

Data mining
Anti-bot services
Enterprise security management
Limiting number of accesses to services

Third-Party Cookies

- Set by **third-party content** (Ads, Social media, etc.)
- Traditionally (ab)used for **cross-site tracking**
- Slowly **dying**
  - Unsuitable for **cross-browser** tracking
  - Users can **delete** them
  - Under **regulation** (GDPR, CCPA)
  - **Browser support is phased-out**

⇒ Trackers are now looking for new tracking technologies

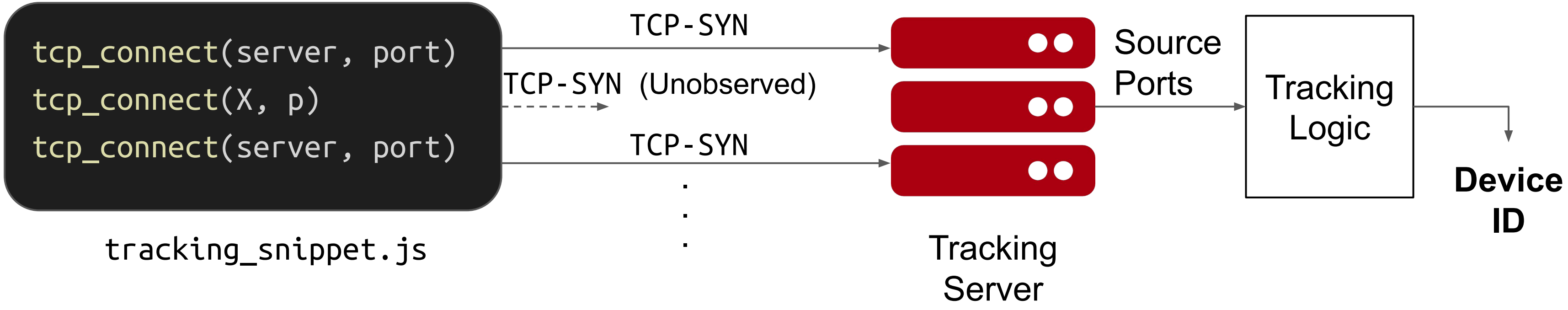
Our Work

- Our work: Uses **TCP source ports** for user tracking
- Entry point: A **tracking snippet** (JavaScript)
- The snippet runs on **the user browser**, executes the **tracking logic** and obtains a **device ID**

Our Work

TCP source ports are not exposed via JavaScript ⇒ **tracking server** is used:

- The **tracking snippet** forces the user device to establish TCP connections
- The **tracking server** analyzes source port measurements for the device ID



```
tcp_connect(server, port)
tcp_connect(X, p)
tcp_connect(server, port)
```

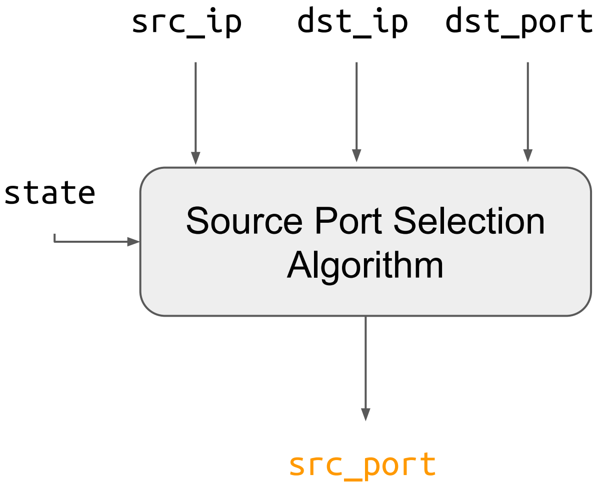
TCP Source Port Selection

- TCP connections are identified by the **4-tuple**:

  `(src_ip, src_port, dst_ip, dst_port)`

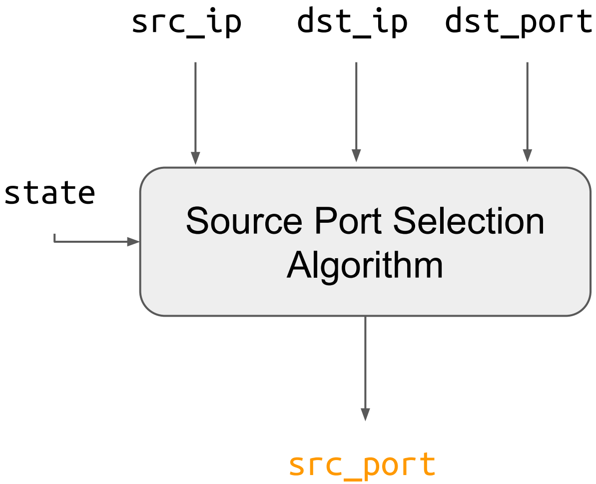
- Client provides `(dst_ip, dst_port)` at **`connect()`**
  - `src_ip` is automatically set by the OS

- OS chooses `src_port` (“Ephemeral Port”)



TCP Source Port Selection

- Source ports should be **unpredictable** and **infrequently reused**
- **RFC 6056** defines 5 port selection algorithms
- Linux moved to the **Double-Hash Port Selection (DHPS) Algorithm** starting from kernel 5.12



RFC 6056, Algorithm 4: Double-Hash Port Selection (DHPS)

- Algorithm state:
    - `table`: Table of 32-bit integers of length T (initialized randomly at boot time)
    - K_1, K_2: Two 128-bit keys (initialized randomly at boot time)
    - `F`, `G`: Two cryptographic keyed hash functions

- To select `src_port` for a given 3-tuple `(src_ip, dst_ip, dst_port)`:
    - Compute `offset = F(`K_1`, src_ip, dst_ip, dst_port)` $\in [0, 2^{32}-1]$
    - Compute `index = G(`K_2`, src_ip, dst_ip, dst_port)` $\in [0, T-1]$
    - Compute `port = min_ephemeral + (table[index] + offset) % num_ephemeral`
    - Increment `table[index]` by 1

RFC 6056, Algorithm 4: Security Issues

- Issue #1: Shared global state
    - Across network interfaces, protocols and network namespaces
- Issue #2: Small table length
    - RFC 6056 suggests `T = 10`; Linux uses `T = 256`
- Issue #3: Deterministic change of state
    - Table cells are incremented by 1

Device Tracking based on DHPS

- Algorithm state is **shared** between **Internet-facing** and **loopback** interfaces

- Loopback has a **fixed** IP address: **127.0.0.1**

```
TCP-Connect(“127.0.0.1”, x)
    ⇒ index = G(K₂, “127.0.0.1”, “127.0.0.1”, x)
```

Device Tracking based on DHPS

- The idea: Collect **hash collisions** of loopback traffic

  i.e. Pairs (**x**, **y**) such that

  `G(K₂, “127.0.0.1”, “127.0.0.1”, x) = G(K₂, “127.0.0.1”, “127.0.0.1”, y)`

- The pairs $\{(x_i, y_i)\}$ are **network independent** (depend only on K_2)

  ⇒ $\{(x_i, y_i)\}$ form a **device ID**.

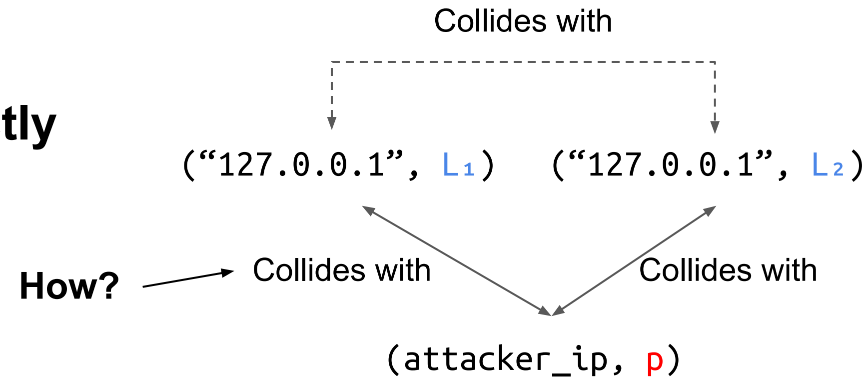
- Device ID persists across browsers, network switches, etc.
  - Lasts as long as K_2 persists (until reboot, in Linux)

Device Tracking based on DHPS

- Challenge: No access to source port information from JavaScript
    - Also, no access to the algorithm state (`table`, K_1, K_2)

- We can only observe the source ports directed to the attacker server

- We will find loopback collisions **indirectly**



Collides with

("127.0.0.1", L_1) ("127.0.0.1", L_2)

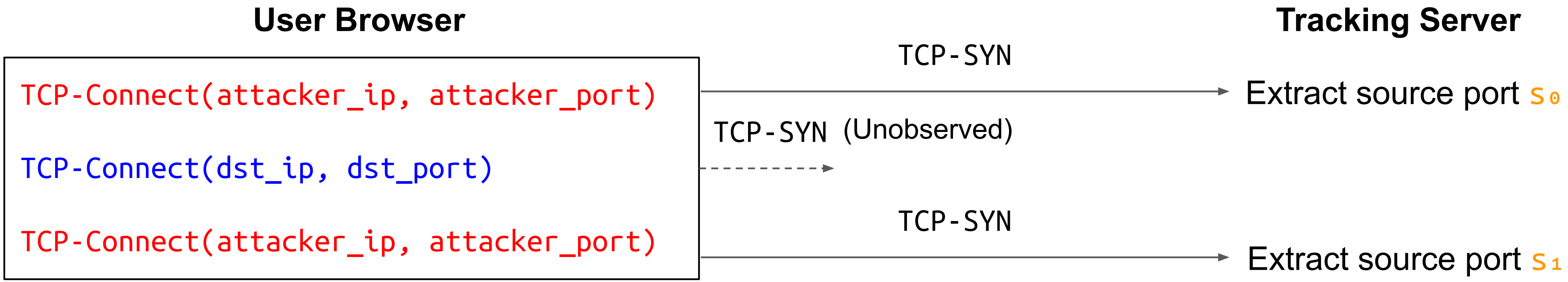
How? → Collides with

Collides with

(attacker_ip, p)

Device Tracking based on DHPS: The Primitive

The user’s device is forced (via JS) to establish 3 TCP connections:



User Browser

Tracking Server

```
TCP-Connect(attacker_ip, attacker_port)

TCP-Connect(dst_ip, dst_port)

TCP-Connect(attacker_ip, attacker_port)
```

Extract source port s_0

Extract source port s_1

Attacker computes the source port difference $\Delta = s_1 - s_0$.

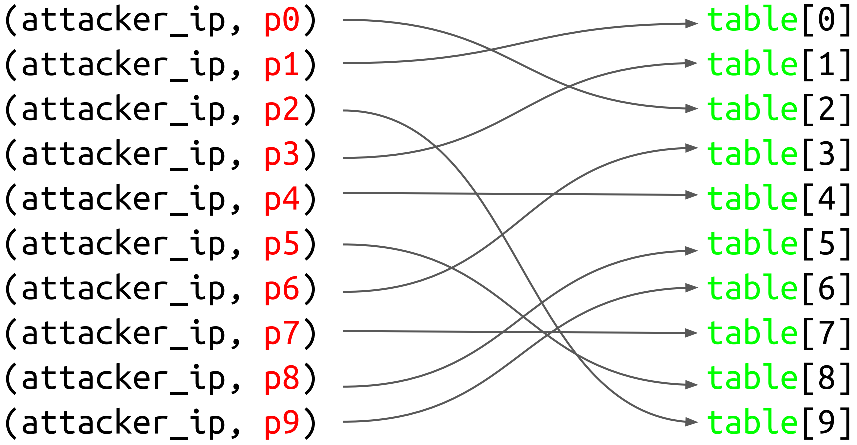
If there was a collision then $\Delta = 2$ (the table cell was incremented twice).

Otherwise, $\Delta = 1$.

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- Phase 1: Collect `T` attacker destinations that uniquely cover all `table` cells

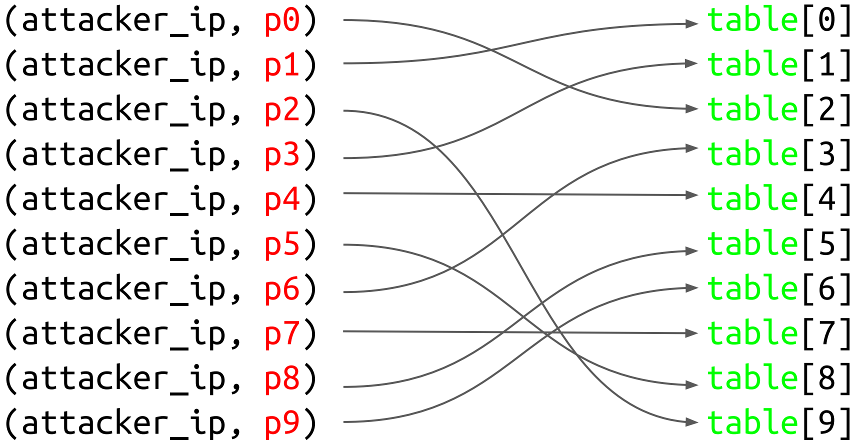


Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 1:</u> Collect `T` attacker destinations that uniquely cover all `table` cells

  The attacker **doesn't know** the mapping, only that it **exists**.



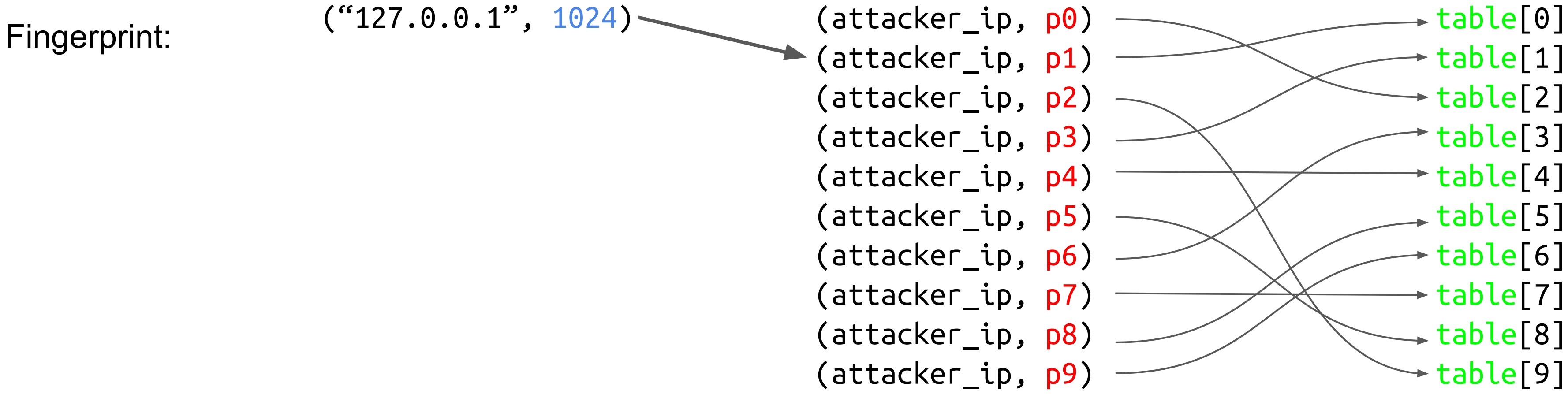
(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)

table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 2:</u> Collect loopback hash collisions



Fingerprint:

`("127.0.0.1", 1024)`

`(attacker_ip, p0)`
`(attacker_ip, p1)`
`(attacker_ip, p2)`
`(attacker_ip, p3)`
`(attacker_ip, p4)`
`(attacker_ip, p5)`
`(attacker_ip, p6)`
`(attacker_ip, p7)`
`(attacker_ip, p8)`
`(attacker_ip, p9)`

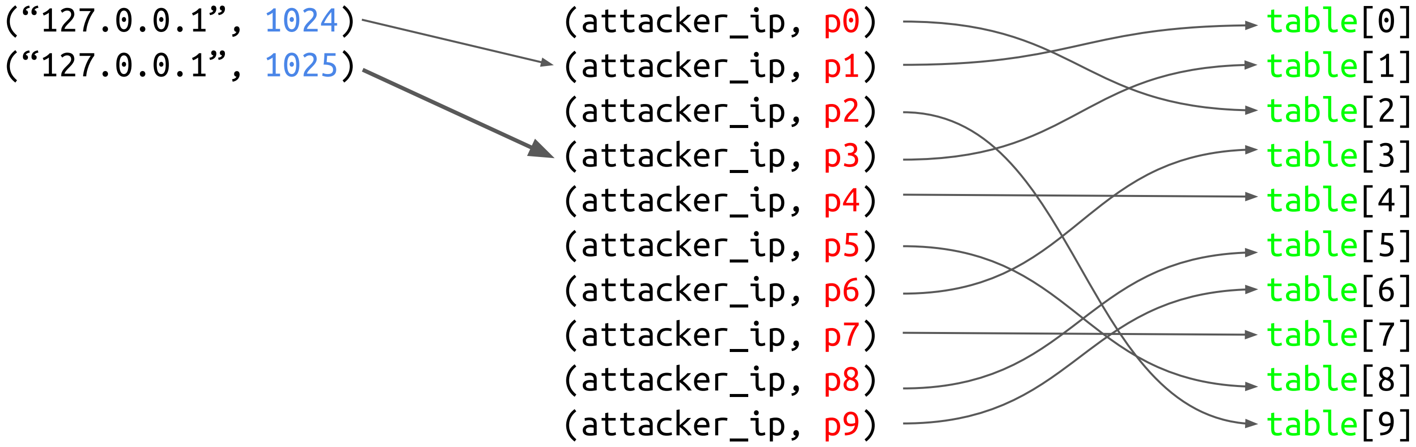
`table[0]`
`table[1]`
`table[2]`
`table[3]`
`table[4]`
`table[5]`
`table[6]`
`table[7]`
`table[8]`
`table[9]`

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 2:</u> Collect loopback hash collisions

Fingerprint:



(“127.0.0.1”, 1024)
(“127.0.0.1”, 1025)

(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)

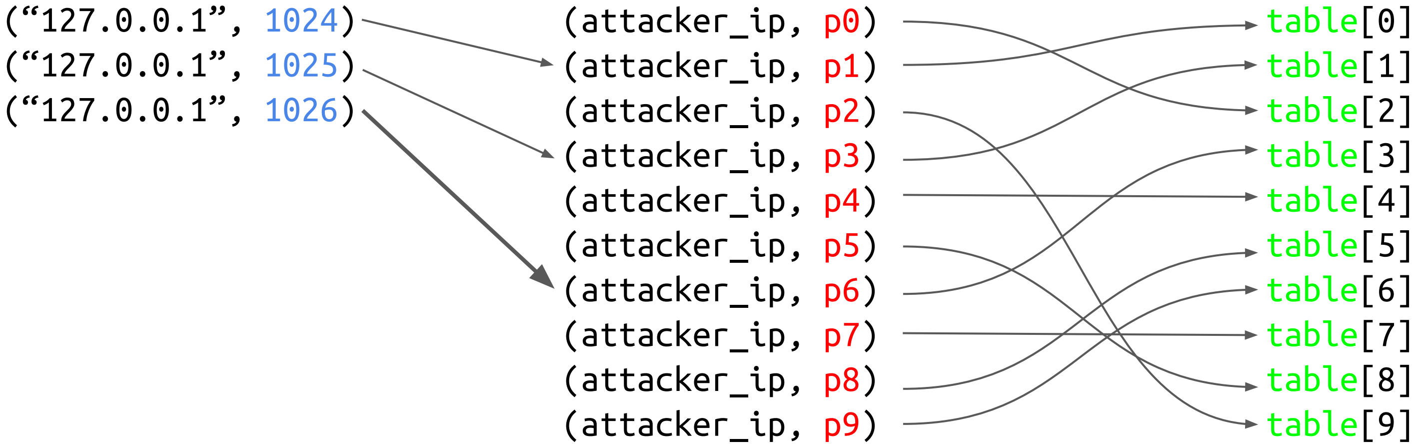
table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 2:</u> Collect loopback hash collisions

Fingerprint:



("127.0.0.1", 1024)
("127.0.0.1", 1025)
("127.0.0.1", 1026)

(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)

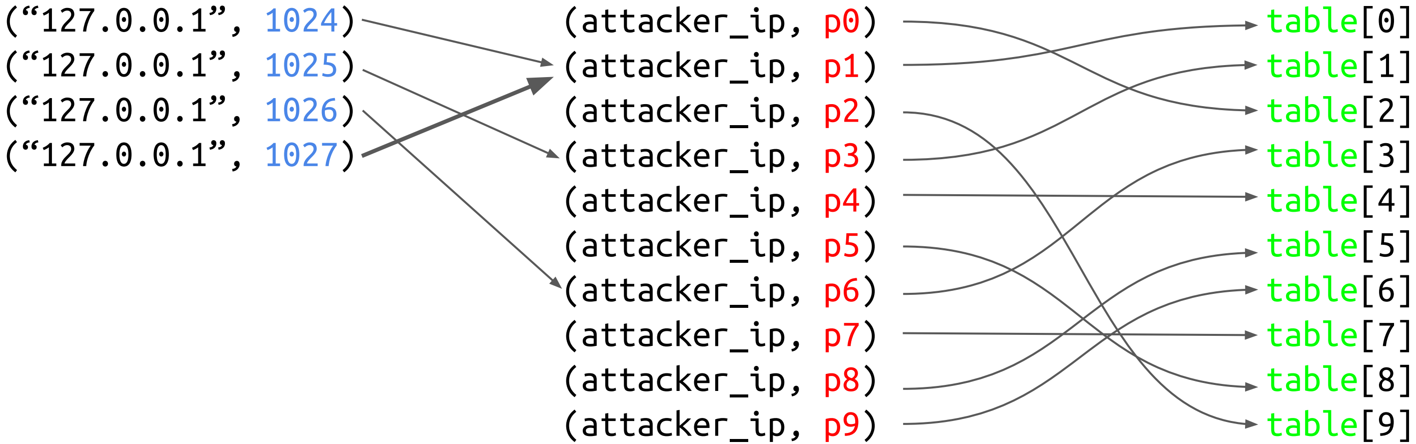
table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- Phase 2: Collect loopback hash collisions

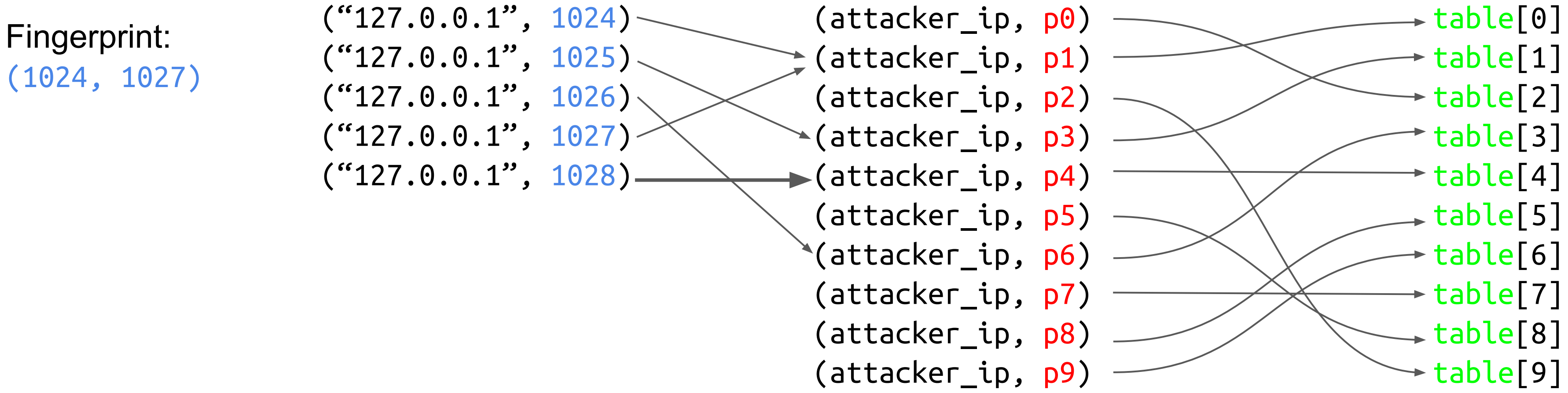
Fingerprint:
`(1024, 1027)`



Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 2:</u> Collect loopback hash collisions



Fingerprint:
`(1024, 1027)`

```
("127.0.0.1", 1024)
("127.0.0.1", 1025)
("127.0.0.1", 1026)
("127.0.0.1", 1027)
("127.0.0.1", 1028)
```

```
(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)
```

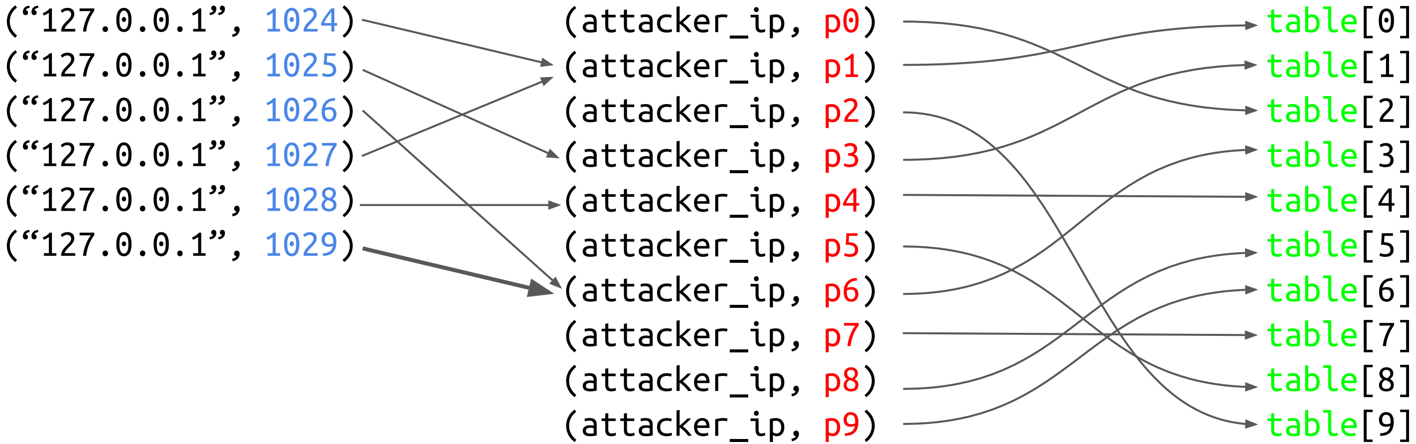
```
table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]
```

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- <u>Phase 2:</u> Collect loopback hash collisions

Fingerprint:
(1024, 1027)
(1026, 1029)



("127.0.0.1", 1024)
("127.0.0.1", 1025)
("127.0.0.1", 1026)
("127.0.0.1", 1027)
("127.0.0.1", 1028)
("127.0.0.1", 1029)

(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)

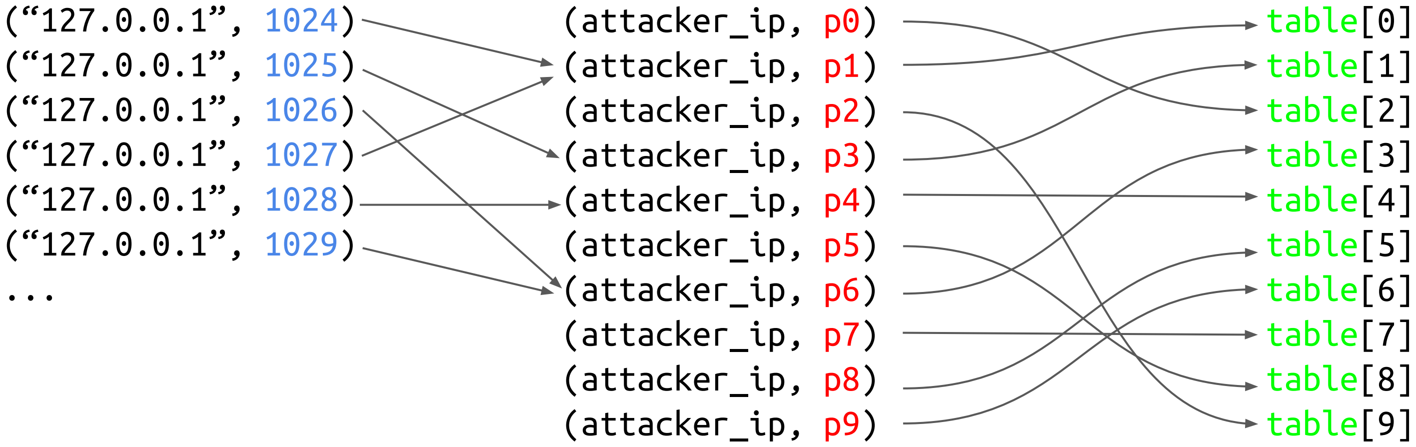
table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]

Device Tracking based on DHPS: The Plan

Our attack works in 2 phases:

- Phase 2: Collect loopback hash collisions

Fingerprint:
(1024, 1027)
(1026, 1029)



(“127.0.0.1”, 1024)
(“127.0.0.1”, 1025)
(“127.0.0.1”, 1026)
(“127.0.0.1”, 1027)
(“127.0.0.1”, 1028)
(“127.0.0.1”, 1029)
...

(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
(attacker_ip, p7)
(attacker_ip, p8)
(attacker_ip, p9)

table[0]
table[1]
table[2]
table[3]
table[4]
table[5]
table[6]
table[7]
table[8]
table[9]

Device Tracking based on DHPS: Phase 1

- The Goal: Obtain `T` attacker destinations that uniquely cover every `table` cell

- The algorithm is **iterative**:
    - `U` ≔ set of **unique** attacker destinations
    - We begin with `U`=∅
    - Expand `U` at every iteration, until `|U|=T` (every table cell is covered)

Device Tracking based on DHPS: Phase 1

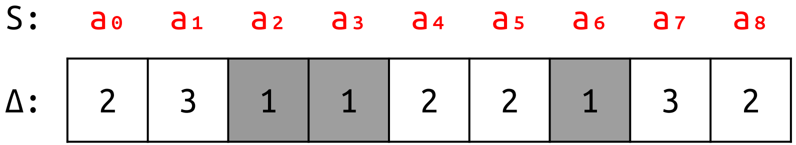
- <u>On each iteration:</u> Extend `U` with more unique attacker destinations.
  - Let `S` be a fresh **set of `T`** random attacker destinations.*
  - Force the user browser to execute (in-order):

    `TCP-Connect(S)`

    `TCP-Connect(U)`

    `TCP-Connect(S)`

  - Update `U` with all `a`∈`S` such that Δ_a==1.



S:	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
Δ:	2	3	1	1	2	2	1	3	2

---

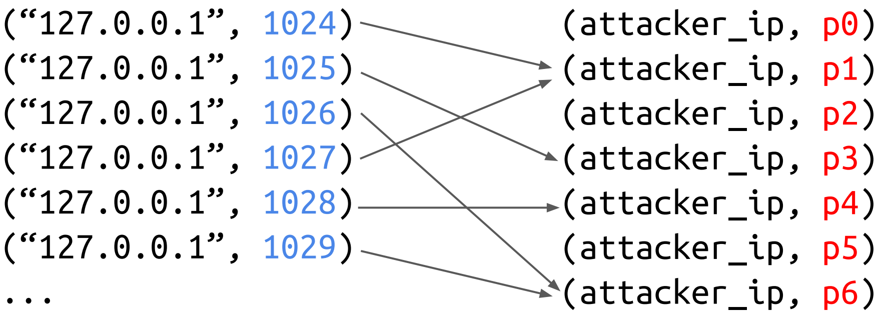
* `T` maximizes the expected number of new unique attacker destinations.

Device Tracking based on DHPS: Phase 2

- <u>The Goal:</u> Collect loopback collisions

- We do this indirectly:
  - We first map loopback destinations to attacker destinations
  - Then, we infer loopback collisions from the mapping

Fingerprint:
`(1024, 1027)`
`(1026, 1029)`
`...`



```
("127.0.0.1", 1024)
("127.0.0.1", 1025)
("127.0.0.1", 1026)
("127.0.0.1", 1027)
("127.0.0.1", 1028)
("127.0.0.1", 1029)
...
```

```
(attacker_ip, p0)
(attacker_ip, p1)
(attacker_ip, p2)
(attacker_ip, p3)
(attacker_ip, p4)
(attacker_ip, p5)
(attacker_ip, p6)
```

Device Tracking based on DHPS: Phase 2

- On each iteration ℓ:

  Find the attacker destination that collides with `("127.0.0.1", ℓ+1024)`:

  ```
  TCP-Connect(U)
  TCP-Connect("127.0.0.1", ℓ+1024)
  TCP-Connect(U)
  ```

  ⇒ Look for a∈U s.t. $\Delta_a > 1$.

U:	u0	u1	u2	u3	u4	u5	u6	u7	u8
Δ:	1	1	1	2	1	1	1	1	1

- We stop when "enough" loopback collisions were collected

We guarantee that: $\Pr[\mathbf{sig}(D) = \mathbf{sig}(D')] \leq \frac{1}{\binom{N}{2}}$

N: population size
D, D': random devices

Device Tracking based on DHPS: Improvements

- Grouping loopbacks: Test α loopbacks instead of one, in phase 2.

```
TCP-Connect(U)
TCP-Connect(“127.0.0.1”, 1024)×$2^0$
TCP-Connect(“127.0.0.1”, 1025)×$2^1$
TCP-Connect(“127.0.0.1”, 1026)×$2^2$
TCP-Connect(“127.0.0.1”, 1027)×$2^3$
TCP-Connect(U)
```

Phase 2 (α=4)

Device Tracking based on DHPS: Improvements

- Robustify against organic noise: Tolerate up to β noise.

```
TCP-Connect(U)
TCP-Connect(“127.0.0.1”, 1024)$\times 2^0 \times \beta$
TCP-Connect(“127.0.0.1”, 1025)$\times 2^1 \times \beta$
TCP-Connect(“127.0.0.1”, 1026)$\times 2^2 \times \beta$
TCP-Connect(“127.0.0.1”, 1027)$\times 2^3 \times \beta$
TCP-Connect(U)
```

Phase 2 (α=4)

Note: Phase 1 already tolerates noise

Device Tracking based on DHPS: Limitations

- Our technique relies on **observing the machine-generated source ports**

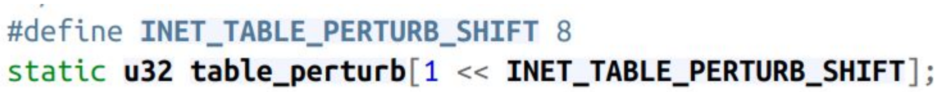
- Cannot track proxy (Tor) users

- Cannot track under networks with port-rewriting NATs

DHPS Implementation in Linux

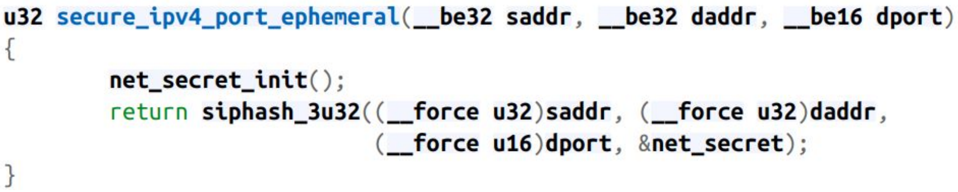
- DHPS is implemented in `__inet_hash_connect()`
    - Used by both IPv4 and IPv6 code

- Table is **global** and its size is **256**



```
#define INET_TABLE_PERTURB_SHIFT 8
static u32 table_perturb[1 << INET_TABLE_PERTURB_SHIFT];
```

- SipHash with 128-bit key for `F, G`

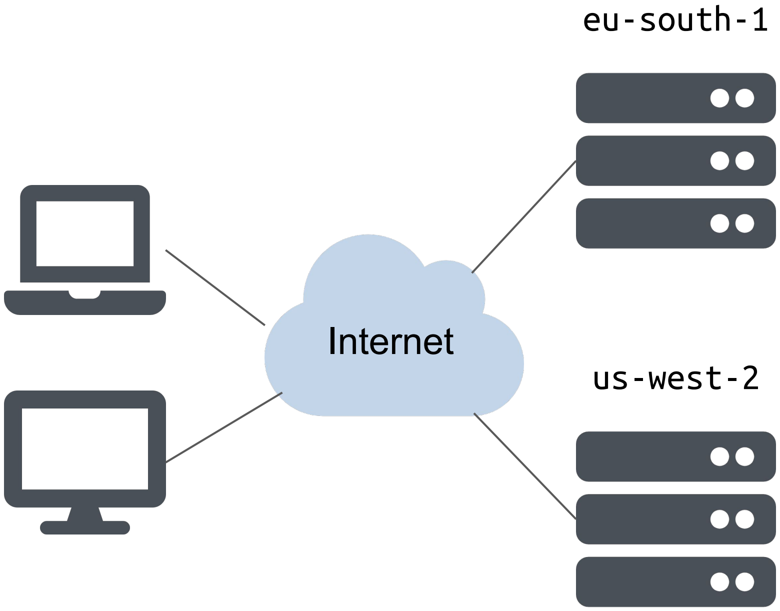


```
u32 secure_ipv4_port_ephemeral(__be32 saddr, __be32 daddr, __be16 dport)
{
        net_secret_init();
        return siphash_3u32((__force u32)saddr, (__force u32)daddr,
                            (__force u16)dport, &net_secret);
}
```

- Noise Injection: Increment a table cell **twice** with probability 1/16

Evaluation: Setup

- Prototype **tracking server** in Go and **tracking snippet** in HTML+JavaScript
- 2 tracking servers in different locations
- Multiple Linux laptops and PCs



Evaluation: Results

- ✓ **Cross browser tracking**
  - Tested Chrome v96.0 and Firefox v96

Evaluation: Results

✓ Cross browser tracking

✓ **Cross browser privacy modes tracking**

Evaluation: Results

- ✓ Cross browser tracking
- ✓ Cross browser privacy modes tracking
- ✓ **Dwell time: 5-15s on Chrome**

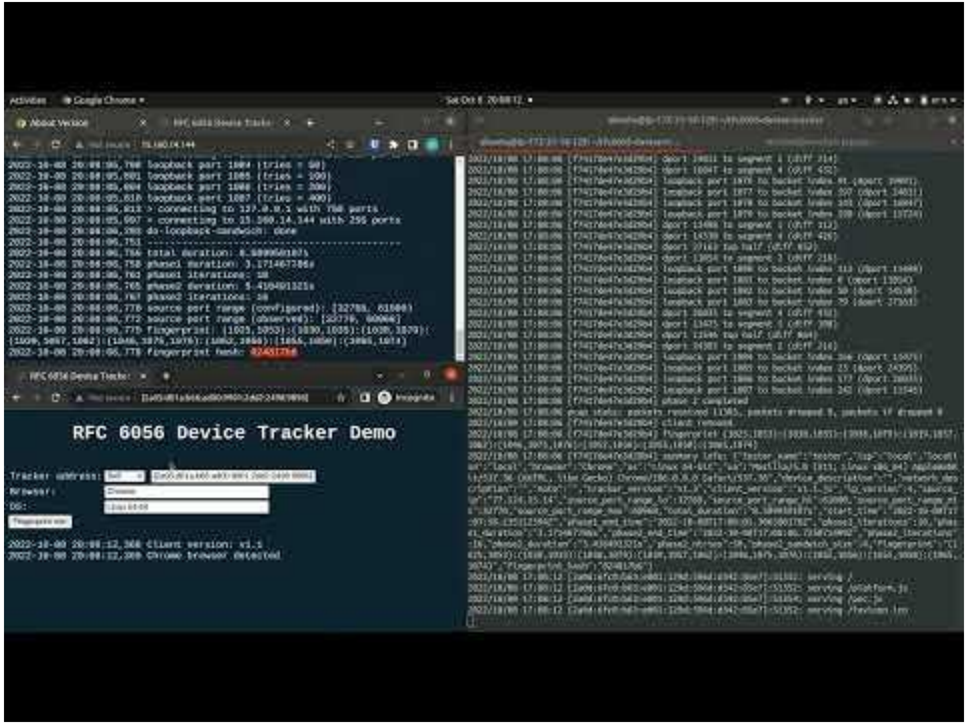
Evaluation: Results

- ✓ Cross browser tracking
- ✓ Cross browser privacy modes tracking
- ✓ Dwell time: 5-15s on Chrome
- ✓ **Cross network tracking**
    - Consistency under IPv4 or IPv6
    - Works on some VPNs: TunnelBear and ExpressVPN

Evaluation: Results

- ✓ Cross browser tracking
- ✓ Cross browser privacy modes tracking
- ✓ Dwell time: 5-15s on Chrome
- ✓ Cross network tracking
- ✓ **Android**
  - Tested Samsung Galaxy S21 with patched kernel
  - Dwell time: 18-21s on Chrome mobile

Demo



Full source code: https://github.com/0xkol/rfc6056-device-tracker

Vendor Status

- February 2022: Linux kernel security team was informed
- Assigned **CVE-2022-32296**
- Worked with the security team to patch the vulnerability
- May 2022: A patch was merged into the Linux kernel

Countermeasures

Countermeasure	**Original Implementation**	**New Implementation**	**Effect**
Increase table size	T=256	T=64K	Collisions become less frequent

Countermeasures

Countermeasure	Original Implementation	New Implementation	Effect
Increase table size	T=256	T=64K	Collisions become less frequent
Periodic re-keying	Every reboot	Every 10s	Collisions become useless after re-keying

Countermeasures

Countermeasure	Original Implementation	New Implementation	Effect
Increase table size	T=256	T=64K	Collisions become less frequent
Periodic re-keying	Every reboot	Every 10s	Collisions become useless after re-keying
More noise	$\sim\mathrm{Bernoulli}(1/16)$	$\sim\mathcal{U}\{0,7\}$	Collisions become harder to determine

Conclusion

- DHPS (Algorithm 4 of RFC 6056) is vulnerable to device tracking
- Demonstrated our technique on Linux
- The device ID persist across browsers, browser privacy modes, networks, etc.
- Shows that user privacy can be undermined in non-obvious ways

Thanks for Listening!

Questions?

Extended paper: https://arxiv.org/pdf/2209.12993.pdf
Source code: https://github.com/0xkol/rfc6056-device-tracker
Demo video: https://www.youtube.com/watch?v=pZbfV5nCQsA