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

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

https://www.usenix.org/conference/usenixsecurity23/presentation/kol

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

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

Moshe Kol
Hebrew University of Jerusalem

Amit Klein
Hebrew University of Jerusalem

Yossi Gilad
Hebrew University of Jerusalem

Abstract

We describe a tracking technique for Linux devices, exploiting a new TCP source port generation mechanism recently introduced to the Linux kernel. This mechanism is based on an algorithm, standardized in RFC 6056, for boosting security by better randomizing port selection. Our technique detects collisions in a hash function used in the said algorithm, based on sampling TCP source ports generated in an attacker-prescribed manner. These hash collisions depend solely on a per-device key, and thus the set of collisions forms a device ID that allows tracking devices across browsers, browser privacy modes, containers, and IPv4/IPv6 networks (including some VPNs). It can distinguish among devices with identical hardware and software, and lasts until the device restarts.

We implemented this technique and then tested it using tracking servers in two different locations and with Linux devices on various networks. We also tested it on an Android device that we patched to introduce the new port selection algorithm. The tracking technique works in real-life conditions, and we report detailed findings about it, including its dwell time, scalability, and success rate in different network types. We worked with the Linux kernel team to mitigate the exploit, resulting in a security patch introduced in May 2022 to the Linux kernel, and we provide recommendations for better securing the port selection algorithm in the paper.

1 Introduction

Online browser-based device tracking is a widespread practice, employed by many Internet websites and advertisers. It allows identifying users across multiple sessions and websites on the Internet. A list of motivations for web-based device tracking (fingerprinting) is listed in [1] and includes "fraud detection, protection against account hijacking, anti-bot and anti-scraping services, enterprise security management, protection against DDOS attacks, real-time targeted marketing, campaign measurement, reaching customers across devices, and limiting the number of accesses to services".

Device tracking is often performed to personalize ads or for surveillance purposes. It can either be done by sites that users visit or by third-party companies (e.g. advertisement networks) which track users across multiple websites and applications ("cross-site tracking"). Traditionally, cross-site tracking was implemented via 3rd party cookies. However, nowadays, users are more aware of the cookies' privacy hazards, and so they use multiple browsers, browser privacy mode, and cookie deletion to avoid such tracking. In addition, support for 3rd party cookies in major browsers is being withdrawn due to privacy concerns [5, 26]. Trackers are, therefore, on the look for new tracking technologies, particularly ones that can work across sites and across browsers and privacy modes, thereby breaking the isolation the latter attempt to provide.

Probably the most alarming impact of device tracking is the degradation of user privacy – when a user's device can be tracked across network changes, different browsers, VPNs, and browser privacy modes. This means that users who browse to one site with some identity (e.g., user account), then browse to another site, from another browser, another network (or VPN), and perhaps at another time altogether, using a completely different and unrelated second identity, may still have the two identities linked.

Often, device tracking techniques are used in a clandestine manner, without the user's awareness and without obtaining the user's explicit consent. This motivates researchers to understand the challenges of device tracking, find new tracking techniques that can be used without consent, and work with the relevant software vendors to eliminate such techniques and raise awareness of these new kinds of attacks.

In this paper, we present a new browser-based tracking technique that supports tracking across IPv4 and IPv6 networks, browsers, VPNs, and browser privacy modes. Our tracking technique can provide up to 128 bits of entropy for the device ID (in the Linux implementation) and requires negligible CPU and RAM resources for its operation. Our technique uses standard web technologies such as Javascript, WebRTC TURN (in Chrome), and XHR (in Firefox). It assumes that a browser renders a web page with an embedded tracking

HTML snippet that communicates with a 1st-party tracking server (i.e., there is no reliance on common infrastructure among the tracking websites). The tracking server then calculates a device ID. This ID is based on kernel data. Therefore, the same device ID is calculated by any site that runs the same logic, regardless of the network from which the tracked device arrives, or the browser used.

The tracking technique is based on observing the TCP source port numbers generated by the device's TCP/IP stack, which is implemented in the operating system kernel. There are several popular TCP source port generation algorithms that differ in the level of security vs. functionality they deliver. Security-wise, TCP source ports should be as unpredictable as possible to off-path attackers [13, §1]. Functionality-wise, TCP source ports should not repeat too often (to mitigate the "instance-id collision" problem [13, §2.3]).

RFC 6056 "Recommendations for Transport-Protocol Port Randomization" [13, §3.3] lists five algorithms used by different operating systems to generate TCP source port numbers. According to RFC 6056, the "Double-Hash Port Selection" algorithm [13, §3.3.4] offers the best trade-off between the design goals of TCP source ports (see § 3), and indeed it was recently adopted with minor modifications by Linux (starting with kernel version 5.12-rc1). Our analysis targets this port selection algorithm, which we expect to propagate into Android devices as well (as Android 13 launches with kernel version 5.15 [19]).

Our technique finds hash collisions in one of the algorithm's hash functions. These collisions depend only on a secret hashing key that the OS kernel creates on boot time and maintains until the system is shut down. Thus, the set of collisions forms a device ID that spans the lifetime of this key, surviving changes to networks, transitions into and out of sleep mode, and using containers on the same machine. It does not rely on the specific choice of the hash functions for the RFC 6056 algorithm beyond the RFC's own requirement, and as such, it presents a generic attack against the RFC algorithm. Since our technique relies on the client's port selection algorithm, it also has some limitations. Specifically, it is ineffective when the client uses Tor or an HTTP forward proxy since they terminate the TCP connection originating at the device and establish their own TCP connection with the tracking server. Furthermore, if a middlebox rewrites the TCP source ports or throttles the traffic, it can interfere with our technique. However, we note that this kind of interference is typically NAT-related and as such is unlikely to apply to IPv6 networks.

We implemented the device tracking technique and tested it with Linux devices across various networks (cellular, WiFi, Ethernet), VPNs, browsers (Chrome, Firefox), browser privacy modes, and Linux containers. It requires the browser to dwell on the web page for 10 seconds on average, which aligns with our theoretical analysis. The resources device tracking requires from the attacker are low, which allows calculating IDs for millions of devices per tracking server. Since off-the-shelf Android devices have not yet deployed Linux kernels that use the new port selection algorithm, we introduced the new algorithm through a patch to a Samsung Galaxy S21 mobile phone (Android device) and tested it. We recommended countermeasures to the Linux kernel team and worked with them on a security patch that was recently released (May 2022). We discuss these recommendations in the paper.

In sum, we make the following contributions:

- Analysis of RFC 6056's "Double-Hash Port Selection" algorithm, showing that a practical device tracking attack can be mounted against devices using this algorithm.
- Adaptation of our device tracking technique to Linux.
- Demonstration and measurements of our device tracking technique across the Internet, in practical settings, under various conditions (browsers, networks, devices, containers, VPNs).
- Full source code of our demonstration tracking server.

2 Related Work

The challenges facing device tracking nowadays revolve around the reliability and scope of the available tracking techniques. Ideally, a web-based tracking technique should work across browser privacy modes (switching between the normal browsing mode and the privacy mode such as "incognito"), across browsers, across networks (switching between cellular networks, WiFi and Ethernet networks), and even across VPN connections. Furthermore, a tracking technique should address the "golden image" challenge [8], wherein an organization provisions a multitude of identical devices (fixed hardware and software configuration) to its employees. The tracking technique should thus tell these devices apart, even though there is no difference in their hardware and software.

Device tracking techniques can be categorized into *tagging* and *fingerprinting* techniques [27]. Tagging techniques insert an ID to the device, typically at the browser level (e.g., a resource cached by the browser or an object added to the standard browser storage such as cookies or `localStorage`). Fingerprinting techniques measure a system or browser's features that can tell devices apart, such as fonts, hardware, and system language.

Klein and Pinkas [8] provide an extensive review of browser-based tracking techniques, current for 2019. They evaluate the techniques' coverage of the golden image challenge and ability to cross the privacy mode gap. They find that typically, fingerprinting techniques fail to address the golden image challenge, while tagging techniques fail when users use the browser in privacy mode. They found that no single existing technique was able to fulfill both requirements. The technique suggested in [8] does not work across networks,

and as such, its practicality is limited. A recent (2020) list of fingerprinting techniques (none of which overcomes the golden image challenge by default) is provided in [12].

Since the analysis provided in [8], the browser-based tracking landscape grew with several new techniques. A tagging technique [24] used the browser favicons cache to store a device ID. This was since fixed in Chrome 91.0.4452.0 [3, 23]. A fingerprinting technique based on measuring GPU features from the WebGL 2.0 API is provided in [6]. A fingerprinting method based on TLS client features and behavior is described in [14]. All the above techniques suffer from their respective category (fingerprinting/tagging) drawbacks.

A technique somewhat similar to [8] that uses the stub resolver DNS cache and times DNS cache miss vs. DNS cache hit events is presented in [15], but this technique (like [8]) does not work across networks.

The "Drawn Apart" browser-based tracking technique [11] is based on measuring timing deviations in GPU execution units. This technique is oblivious to the device's network configuration and the choice of browser and privacy mode and can also tell apart devices with identical hardware and software configurations. However, it does so with limited accuracy – 36.7%-92.7% in lab conditions [11, Table 1], which is insufficient for large-scale tracking.

Klein et al.'s works [2, 7, 9] revolved around a tracking concept based on kernel data leakage, which identifies individual devices. The leakage occurred in various network protocol headers (IPv4 ID, IPv6 flow label, UDP source port). All of them were quickly fixed by the respective operating system vendors due to their severity and impact and were no longer in effect when our research was conducted.

3 Background

RFC 6056 [13, Section 3.3] analyzes five TCP source port allocation algorithms and defines several design goals [13, Section 3.1]. The two goals relevant to this work are:

- Minimizing the predictability of the ephemeral port numbers used for future transport-protocol instances.
- Minimizing collisions of instance-ids [TCP 4-tuples].

The first goal aims for security against blind TCP attacks, such as blind reset or data injection attacks [20]. The second goal is functionality-related and ensures that successive port assignments for the same destination (IP and port) do not re-use the same source port since this can cause a TCP failure at the remote end. That is because if the device terminates one TCP connection, the server may still keep it active (in the TCP TIME_WAIT state) while the device attempts to establish a second connection with the same TCP 4-tuple, which will fail since the server has this 4-tuple still in use.

Double-Hash Port Selection Algorithm. Our focus is on RFC 6056's Algorithm 4, "Double-Hash Port Selection Algorithm" (DHPS), which is detailed in Algorithm 1. This algorithm selects a TCP source port for a given $IP_{SRC}, IP_{DST}, PORT_{DST}$, which we term the connection's *3-tuple*. Thus the algorithm completes a 3-tuple into a (TCP) 4-tuple. In this algorithm, *table* is a "perturbation table" of T integer counters (in the Linux kernel, $T = 256$), F is a cryptographic keyed-hash function which maps its inputs to a large range of integers, e.g., $[0, 2^{32} - 1]$, and G is a cryptographic keyed-hash function which maps its inputs to $[0, T - 1]$. The TCP source ports the algorithm produces are in the range $[min\_ephemeral, max\_ephemeral]$ (in the Linux kernel, by default, $min\_ephemeral = 32768, max\_ephemeral = 60999$). DHPS calculates an index i to a counter in *table* based on a keyed-hash (G) of the given 3-tuple and uses the counter value offset by another keyed-hash (F) of the 3-tuple as a first candidate for a port number (using modular arithmetic to produce a value in $[min\_ephemeral, max\_ephemeral]$). DHPS then checks whether the port number candidate is suitable (CHECKSUITABLEPORT). This check is intentionally underspecified in the RFC so that each implementation may run its own logic. For example, the Linux kernel checks whether there already is a 4-tuple with these parameters. If this check passes, DHPS returns the candidate port; otherwise, it increments the candidate port and runs the check again.

Algorithm 1 DHPS Source Port Selection (RFC 6056 §3.3.4)

```
1:  procedure SELECTEPHEMERALPORT
2:      num_ephemeral ←
3:          max_ephemeral − min_ephemeral + 1
4:      offset ← F_K1(IP_SRC, IP_DST, PORT_DST)
5:      index ← G_K2(IP_SRC, IP_DST, PORT_DST)
6:      count ← num_ephemeral
7:      repeat
8:          port ← min_ephemeral+
9:              ((offset + table_index) mod num_ephemeral)
10:         table_index ← table_index + 1
11:         if CHECKSUITABLEPORT(port) then
12:             return port
13:         count ← count − 1
14:     until count = 0
15:     return ERROR
```

Port selection in the Linux kernel. Linux version 5.12-rc1 switched from RFC 6056's Algorithm 3 ("Simple Hash-Based Port Selection Algorithm") to DHPS, quoting security and privacy concerns as the reason for this change [4]. Starting from this version, the Linux kernel uses DHPS to generate TCP source ports for outbound TCP connections over IPv4 and IPv6. The Linux implementation and its few minor modifications are discussed in § 5.1. Linux kernel version 5.15 is the first long-term service (LTS) kernel version in which DHPS is used, thus LTS Linux installations using unpatched kernel 5.15 and above are vulnerable (see § 9).

Port selection in Android. The Android operating system kernel is based on Linux and as such vulnerabilities found in Linux may also impact Android. The TCP source port selection algorithm in Android depends on the underlying Linux kernel version: devices running Linux kernels ≤ 5.10 do not use DHPS and therefore are not vulnerable, whereas devices running a more recent kernel use DHPS and *may* be vulnerable if unpatched. At the time of writing, Android devices on the market use kernel version 5.10, even for Android 13, though Android 13 running kernel 5.15 is likely to be released in the near future [19]. To assess the feasibility of our attack on Android, we conducted an experiment using an Android device with a modified kernel that includes the flaw (see § 6.5 for results). Since the vulnerability was patched on Linux in May 2022, and the patch was merged into Android as well, we expect future Linux and Android devices that use DHPS to be safe.

4 Device Tracking Based on DHPS

Attack model. We assume the victim's (Linux-based) device runs a browser that renders a web page containing a "tracking snippet" – a small piece of HTML and Javascript code (which runs in the browser's Javascript sandbox). The snippet implements the client-side logic of the tracking technique. It can be embedded in web pages served by, e.g., commerce websites or 3rd-party advertisements. When a device visits these pages, the tracking server logic calculates a unique ID for that device, allowing tracking it both with respect to the time dimension and the space dimension (visited websites).

Device ID. When the browser renders the tracking snippet and executes the Javascript code in it, the code makes the browser engage in a series of TCP connection attempts with the attacker's tracking server, interleaved with TCP connection attempts to a localhost address. By observing the browser's traffic TCP source ports at the tracking server, the attacker can deduce hash collisions on G_{K_2} (this is explained later). The attack concludes when the attacker collects enough pairs of hash collisions on G_{K_2} where IP_{SRC}, IP_{DST} are fixed loopback addresses, i.e., pairs (x,y) such that

$$G_{K_2}(IP_{SRC} = 127.0.0.1, IP_{DST} = 127.1.2.3, PORT_{DST} = x) = G_{K_2}(IP_{SRC} = 127.0.0.1, IP_{DST} = 127.1.2.3, PORT_{DST} = y)$$

These pairs depend only on K_2, and as such, they represent information on K_2 and on it alone. K_2 is statistically unique per device (up to the birthday paradox) and does not rely on the current browser, network, or container. Thus, the set of collisions $\{(x_i, y_i)\}$ forms a device ID that persists across browsers, browser privacy mode, network switching, some VPNs, and even across Linux containers. The ID is invalidated only when the device reboots or shuts down.

The use of loopback addresses is critical to the effectiveness of the attack. Using the Internet-facing address of the device does not yield a consistent device ID across networks. While hash collisions based on the Internet-facing address *can* be calculated, they are of no use when the device moves across networks because the device typically obtains a new, different Internet-facing address whenever it connects to another network. Thus, the collisions calculated for different networks will likely be completely different sets.

Limitations. Our technique tracks client devices through their source port choice. A middlebox such as a NAT may modify the client's source port selection and cause the tracking service to fail to compute a consistent device ID. Furthermore, a device establishing organic TCP connections during the (short) time the attack executes may also thwart the attack; however, we integrate a mechanism for robustness against organic TCP connections. In § 6 we evaluate the attack under these conditions (devices connected through NATs and establishing organic connections while the attack executes). Our technique cannot track clients that connect via forward proxies, which establish a new TCP connection to the tracking server (instead of a direct connection from the client). Particularly, it is ineffective against Tor clients.

4.1 Attack Overview

The attack exploits a core vulnerability in DHPS. DHPS assigns a source port to a destination (IP address and port – 2^{48} combinations for IPv4) using a state maintained in one of the cells of its small perturbation table. This means that many destinations are generated using the same table cell, i.e., using a state that changes in a predictable way between accesses (DHPS increments the cell per each usage). The attack exploits this behavior for detecting collisions in the cell assignment hash function. Such collisions among loopback destinations are invariant to the network configuration of the device and can thus serve as a stable device ID.

To describe the attack, we first define two subsets of tuples that are of special interest for our tracking technique:

- An *attacker 3-tuple* is a 3-tuple in which IP_{DST} is the attacker's tracking server IP address, and IP_{SRC} is the Internet-facing address used by the measured device.
- A *loopback 3-tuple* is a 3-tuple in which IP_{DST} is a fixed loopback address (e.g., 127.1.2.3), and IP_{SRC} is a loopback-facing address used by the measured device (typically 127.0.0.1).

The goal of the attack is to find collisions in G_{K_2} for loopback 3-tuples. These collisions, described as pairs of loopback 3-tuples that hash to the same value, form the device ID.

The attack consists of two phases. In the first phase (Algorithm 2), the attacker obtains T attacker 3-tuples, each one corresponding to one cell of the perturbation table. The attacker does not know which 3-tuple maps to which cell, but that is immaterial to the attack. All the attacker needs is the

Algorithm 2 Finding Attacker 3-Tuple per Cell (Phase 1)

```
1:  procedure SendBurst(X)
2:      for all x ∈ X do
3:          AttemptConnectTCP(x)
4:  procedure GetSourcePorts(U)
5:      SendBurst(U)
6:      R ← ReceiveAttackerTupleToPortMap()
7:          ▷ R = {(IP_SRC, IP_DST, PORT_DST) ↦ PORT_SRC}
8:                  ▷ (obtained from the tracking server)
9:      return R
10: procedure Phase1
11:     S' ← ∅
12:     while |S'| < T do
13:         S_i ← GetNewExternalDestinations()
14:                                   ▷ ∀_{j<i}(S_i ∩ S_j = ∅)
15:         P ← GetSourcePorts(S_i)              ▷ 1st burst
16:         SendBurst(S')                        ▷ 2nd burst
17:         P' ← GetSourcePorts(S_i)             ▷ 3rd burst
18:         S' ← S' ∪ {x | P'(x) − P(x) = 1}
19:                              ▷ V_i = {x | P'(x) − P(x) = 1}
20:     return S'
```

existence of a 1-to-1 mapping between the perturbation table and the T attacker 3-tuples. In the second phase, the attacker maps loopback 3-tuples into attacker 3-tuples (each loopback 3-tuple considered is mapped to the attacker 3-tuple that falls into the same perturbation table cell). This allows the attacker to detect collisions in G_{K_2} for loopback 3-tuples.

4.2 Phase 1

In this phase, the attacker obtains T attacker 3-tuples so that each one corresponds to a unique cell in the perturbation table. This is done in iterations, as shown in Algorithm 2. Define $S'_0 = \varnothing$. In iteration i, the attacker generates a set S_i of *new* attacker destinations (in [10, §A] we show that $|S_i| = T - 1$ minimizes the number of phase 1 iterations). The attacker then instructs the browser to send three bursts of TCP connection attempts (TCP SYN packets): the first burst to S_i, then the second burst to S'_{i-1}, and the third burst to S_i again. An attacker 3-tuple in S_i is determined to be *unique* if the difference between the two sampled source ports for that 3-tuple is 1, indicating that no other attacker 3-tuple in S'_{i-1} or S_i shares the same perturbation table cell. Define V_i to be all such attacker 3-tuples in S_i, and define $S'_i = S'_{i-1} \cup V_i$. This is repeated until $|S'_i| = T$, i.e., all perturbation table cells are uniquely covered by the attacker 3-tuples in S'_i. Figure 1 illustrates a single iteration.

In Appendix A, we also show how phase 1 by itself can be used to measure the rate of outbound TCP connections.

Algorithm 3 Finding a Device ID (Phase 2)

```
1:  procedure Phase2
2:      C ← ∅  ;  n ← 0  ;  i ← 0
3:      repeat
4:          i ← i + 1
5:          P ← GetSourcePorts(S')               ▷ 1st burst
6:          AttemptConnectTCP({L_i})
7:          P' ← GetSourcePorts(S')              ▷ 2nd burst
8:          w ← ɯx(P'_x − P_x > 1)               ▷ |{x | P'_x − P_x > 1}| = 1
9:          if Defined(B_w) then                 ▷ A collision was found
10:             C ← C ∪ {(L_i, B_w)}
11:                 ▷ Add the (single) independent pair to C
12:             n ← n + 1
13:         else
14:             B_w ← L_i
15:     until n ≥ n*_i
16:                 ▷ This is equiv. to P^i_D(n) ≤ p* ( [10, §A])
17:     l ← i
18:     return (C, l)
```

4.3 Phase 2

In the second phase, the attacker goes over a list L of loopback 3-tuples. For each loopback 3-tuple, the attacker finds which attacker 3-tuple (one of the T attacker 3-tuples found in phase 1) belongs to the same perturbation table cell. This mapping allows the attacker to find collisions in G_{K_2} outputs among loopback 3-tuples, which (together with the number of iterations l) form the device ID.

In this phase (Algorithm 3), the attacker repeatedly runs iterations, until enough G_{K_2} collisions are collected. In each iteration, the attacker maps a new loopback 3-tuple L_i to an attacker 3-tuple w, which hashes into the same cell of the perturbation table as the loopback 3-tuple. This is done by "sandwiching" a few loopback 3-tuple L_i packets between bursts to all T attacker 3-tuples obtained in phase 1, and observing which attacker 3-tuple w has a port increment > 1 (see Figure 2). The attacker then collects new collisions with the first loopback 3-tuple in the cell (B_w) and counts the total number of collisions in n. This is illustrated in Figure 3.

The attacker collects and counts "independent" colliding pairs. By this term we mean that if there are exactly k loopback 3-tuples $x_1, \ldots, x_k$ that fall into the same cell, the attacker only uses $k-1$ pairs e.g. $(x_1, x_2), \ldots, (x_1, x_k)$, out of the possible $\binom{k}{2}$ pairs.

Note that our attack makes no assumptions on the choice of Algorithm 1's parameters (the hash functions F, G), beyond assuming that G is reasonably uniform, which is guaranteed since RFC 6056 [13, Section 3.3.4] mandates that "G() should be a cryptographic hash function".

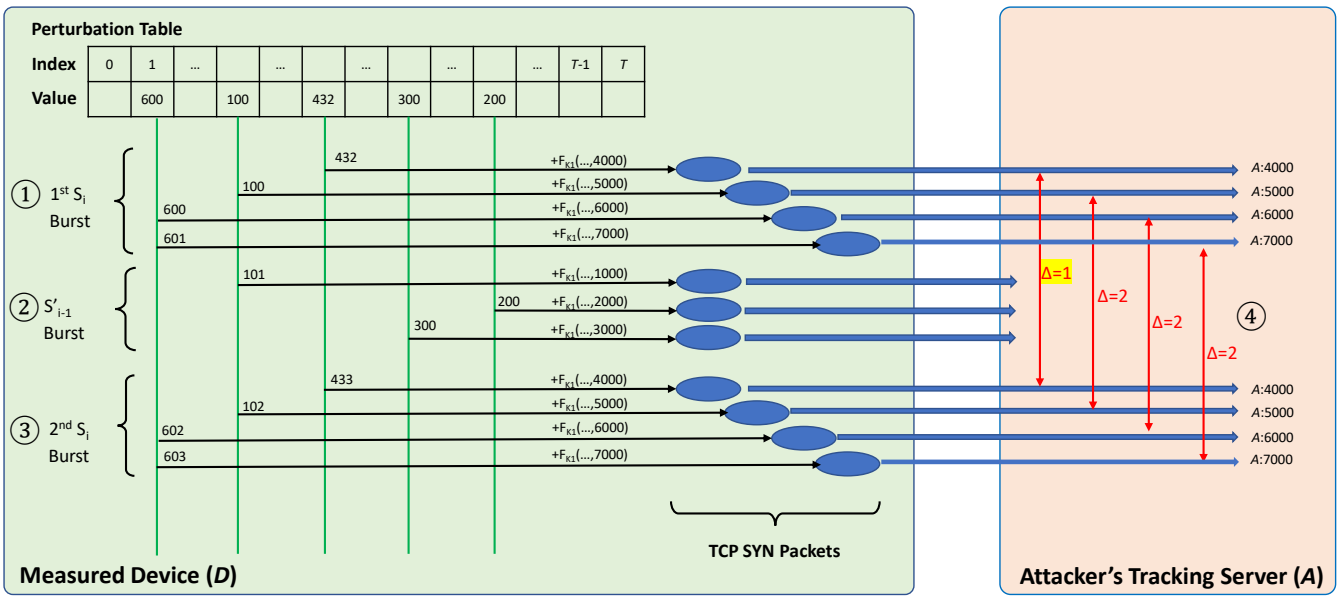

Figure 1: Phase 1 – Single Iteration. This example illustrates how the attacker adds 3-tuples which fall uniquely into cells. In Step (1), the device sends a first burst of TCP SYN packets for the new 3-tuple candidates, S_i (4 in this example). In Step (2), the device sends the burst of TCP SYN packets for the set S'_{i-1} of unique-cell attacker tuples (3 are shown in the illustration). In Step (3), the device sends a second burst of TCP SYN packets for the new 3-tuple candidates. In Step (4), the tracking server detects that only for the attacker 3-tuple which has destination port 4000, the source port was advanced by 1 (yellow background), which indicates that this 3-tuple has a unique cell. The attacker's 3-tuple with destination port 5000 had its source port advanced by 2 because it shares a cell with destination port 1000 in S'_{i-1}, and the 3-tuples with destination ports 6000 and 7000 share a cell; hence their source ports were advanced by 2.

4.3.1 Terminating with an Accurate ID

We want phase 2 to terminate as soon as "enough" collisions are observed. By this, we mean that the probability of another device (i.e., a device with a random K_2) to produce the same set of collisions is below a given threshold. Thus, we define our target function $P_D^l(n)$ to be the probability of a random device getting the same ID as the device D at hand, which is assumed to terminate after l iterations with exactly n independent pairs. (We show below that $P_D^l(n)$ does not depend on the "structure" of the independent collisions, only on their number, i.e., it is well defined.) Note that $P_D^l(n)$ is defined for $l \geq 1$ and $\max(0, l-T) \leq n \leq l-1$. We can define $P_D = 0$ elsewhere. We will then require $P_D^l(n) \leq p^*$, where p^* is a threshold acceptance probability. As explained in [10, §A], the choice of p^* depends on the expected device population size N, e.g. $p^* = 1/\binom{N}{2}$ guarantees there will be up to one ID collision on average in the entire population. We provide p^* values for example population sizes in [10, §A].

To calculate $P_D^l(n)$, we begin by calculating the probability for a random device to have exactly the same set of collisions as device D, after l iterations. Lemma 4.1 shows that this probability is in fact $P_D^l(n)$. Analyzing this probability is a "balls in buckets" problem. The buckets are the T perturbation table cells (that map 1-to-1 to the T attacker 3-tuples from phase 1), and the balls (in iteration l of phase 2) are the loopback 3-tuples that are mapped to the perturbation table cells. However, when looking at two devices, D and D', the attacker has no way to map the buckets between the devices. The only information the attacker can consider is collisions among the balls, i.e., which balls (loopback 3-tuples) fall into the same bucket in both devices. We call this the "structure" of collisions. More formally, we define the structure of collisions in D (after l iterations) as follows: in a device D, we have $l-n$ occupied (non-empty) buckets B_i, and we have a set C_i of loopback 3-tuples in B_i such that $|C_i| = m_i$, and of course $\sum_i m_i = l$. The collision structure in D then is the set of sets $\{C_1, \ldots, C_{l-n}\}$.

For two devices D' and D to have the same structure, we look at the *first* loopback 3-tuple in each bucket – denote f_i as the first loopback 3-tuple in bucket B_i. When building the bucket for D', we start with all-empty buckets. The first loopback 3-tuple f'_1 in B'_1 can pick any bucket, thus the probability to succeed in matching the structure of D is $\frac{T}{T} = 1$. The first loopback 3-tuple f'_2 in B'_2 has probability $\frac{T-1}{T}$ to match D's structure, since it must not hit the first bucket B'_1, and so forth. So the combined probability of all first loopback 3-tuples in their buckets to match D's structure is $\prod_{i=0}^{l-n-1}(1-\frac{i}{T})$. The remaining loopback 3-tuples must each go to its bucket in

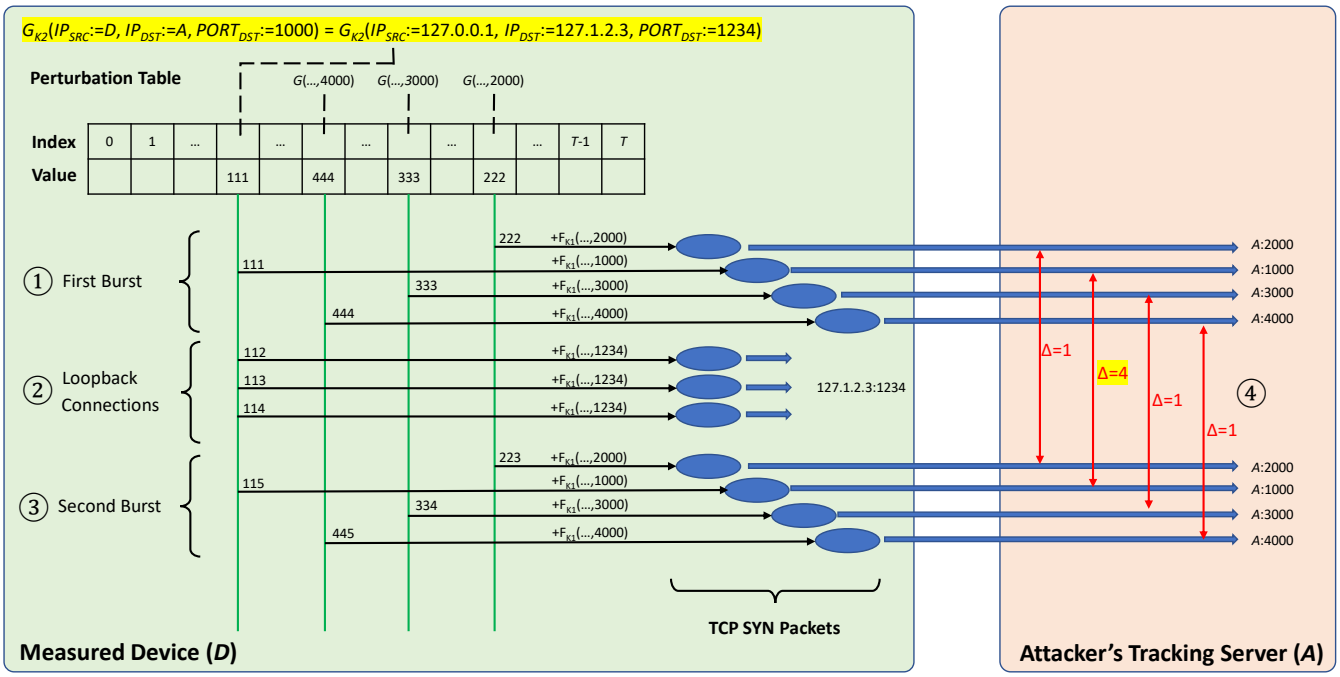

Figure 2: Phase 2 – Single Iteration. This example illustrates how the attacker discovers that the the cell of the loopback 3-tuple $(IP_{SRC} = 127.0.0.1, IP_{DST} = 127.1.2.3, PORT_{DST} = 1234)$ is identical to the cell of the attacker 3-tuple $(IP_{SRC} = A, IP_{DST} = D, PORT_{DST} = 1000)$. In Step ①, the device sends a first burst of TCP SYN packets for all T unique-cell attacker tuples (only 4 are shown in the illustration). In Step ②, the device sends several (3 in this example) TCP SYN packets to the loopback destination 127.1.2.3:1234. In Step ③, the device sends a second burst of TCP SYN packets for all T unique-cell attacker 3-tuples. In Step ④, the tracking server detects that only for the attacker 3-tuple which has destination port 1000, the source port was advanced by at least 4 (right hand side, yellow background), which indicates that this 3-tuple shares the same counter (cell) with the tested loopback 3-tuple (yellow background equation at the upper left corner).

order to match, so each one has probability $\frac{1}{T}$. Therefore,

$$P_D^l(n) = \frac{\prod_{i=0}^{l-n-1}(1-\frac{i}{T})}{T^n}$$

Interestingly, this probability does not depend on $(m_1, m_2, \ldots)$ of the structure – it only depends on the total number of independent collisions, n. This means that $P_D^l(n)$ is well defined.

We now show that $P_D^l(n)$ that we just calculated describes the probability for a random device to have the same device ID as D. This is proved by this lemma:

Lemma 4.1. *If device D' has the same collision structure as in the device ID of device D after l iterations, where l is taken from the signature of D, then D' and D must have an identical device ID.*

Proof. For device D' to have the same device ID as device D, we also need the number of iterations to match – i.e., we need to show that $l' = l$. Recall that the phase 2 algorithm stops as soon as it reaches an iteration l that fulfills the condition $P_D^l(n) \leq p^*$. Also, the order of loopback 3-tuples the algorithm tests is deterministic. Therefore, the algorithm for D' will stop at exactly the same number of iterations as D. □

4.4 Further Improvements

Phase 1 and 2 dwell time optimization. The phase 1 algorithm, as depicted in Algorithm 2, is carried out entirely on the client-side. However, the client cannot obtain the TCP source ports of the TCP connections it attempts to establish with the tracking server (JavaScript code running in the browser has no way of accessing TCP connection information). This information is only available at the tracking server. Thus, the client and the server need to engage in a "ping-pong" of information exchange. This exchange is depicted in procedure GETSOURCEPORTS of Algorithm 2, where the client attempts to establish multiple TCP connections with the server, and the server returns a map from destinations to their TCP source ports. This ping-pong does not need to follow the exact form of Algorithm 2. In particular, there is no need to collect the TCP source ports of the first burst before sending the second burst. The two bursts can be sent one after another without pausing, as long as the client-side (OS) sends the bursts in the order the client prescribes. Similarly, the third burst can be sent by the client immediately after the second burst, since there is no need to synchronize with the server

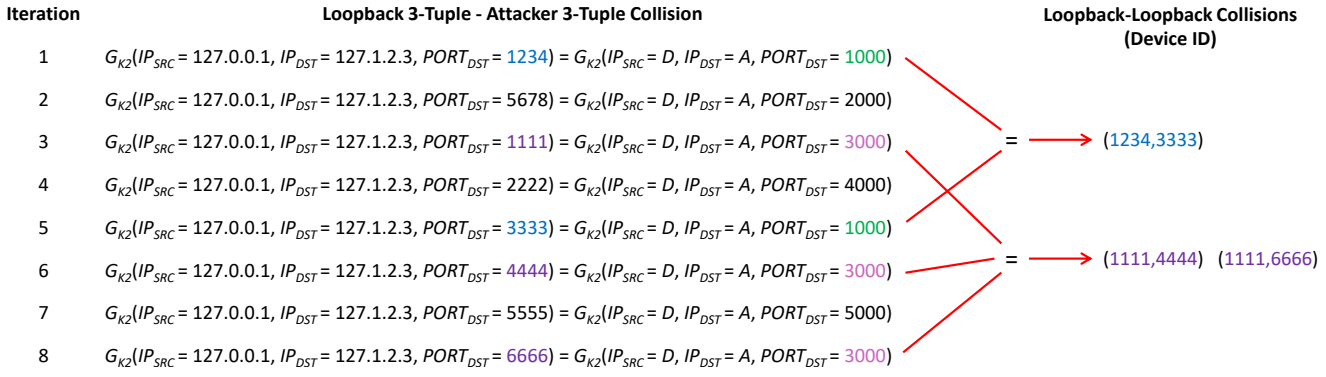

Figure 3: Phase 2 – Calculating a Device ID from Multiple Iterations

after the second burst is emitted. Note that in Algorithm 2, proceeding to the next iteration requires the client to know which destination addresses to add to S' (the set of unique attacker 3-tuples). Thus, moving *across iterations* requires the client and server to synchronize.

The same argument can be applied to phase 2 (Algorithm 3). In essence, this algorithm can run almost asynchronously between the client and the server. The client only ensures that the bursts are distinguishable at the server-side (see below). We further optimize by moving the termination logic to the server and having the client run through the iterations over L_i and only carry out TCP connection attempts. Thus, for each new iteration, the server collects the source ports and updates the attacker 3-tuple cell (B_w), the device ID (C), and the number of independent pairs (n). It also checks the termination condition and signals the client to stop when it is met. In this scheme, the client does not need to wait for the server's response after each burst. Hence, the client is free to make a significant optimization, entirely getting rid of the first burst in each iteration, since the server can use the previous "second" burst just as accurately. The phase 2 algorithm then becomes Algorithm 4.

Algorithm 4 Finding a Device ID (Phase 2, Optimized)

```
1:  procedure PHASE2-SERVER
2:      C ← ∅ ; n ← 0 ; i ← 0
3:      P ← COLLECTSOURCEPORTS(S')
4:      repeat
5:          i ← i + 1
6:          P' ← COLLECTSOURCEPORTS(S')
7:          w ← ɯx(P'_x − P_x > 1)        ▷ |{x|P'_x − P_x > 1}| = 1
8:          if DEFINED(B_w) then    ▷ A collision was found
9:                  ▷ Add the (single) independent pair to C
10:             C ← C ∪ {(L_i, B_w)}
11:             n ← n + 1
12:         else
13:             B_w ← L_i
14:         P ← P'
15:     until n ≥ n*_i
16:             ▷ This is equiv. to P^i_D(n) ≤ p* ( [10, §A])
17:     SIGNALCLIENTSTOP()
18:     l ← i
19:     return (C, l)
20: procedure PHASE2-CLIENT
21:     SENDBURST(S')
22:     for i = 1 to l_max do
23:         ATTEMPTCONNECTTCP(L_i)
24:         SENDBURST(S')
```

Burst separation. The optimization just described mandates the server to separate the traffic back into bursts since packets might be re-ordered by the network. Now that the client and server are not synchronized, this task is a bit more challenging. The client controls the timing on its end, i.e., a burst is followed by loopback traffic, followed by another burst, etc., precisely in this order. On the server-side, we can separate bursts using the source port itself. When we look at all the traffic to a certain destination (port), we can deduce the order of sending by sorting the packets by their TCP source port, which is monotonously increasing with the sending time (by the properties of the DHPS algorithm). So the packet with the lowest TCP source port (for a specific server port destination) belongs to the first burst, the second packet – to the second burst, etc. That being said, source ports may wraparound if $offset + table_{index}$ exceeds *num_ephemeral*. To account for that, we begin by normal sorting and then find the wraparound point by looking for a large enough difference that has been created as a result of the wraparound.

Minimizing number of bursts. We modify the phase 2 algorithm to group several loopback 3-tuples together, sandwiched between attacker-directed bursts. Let α be the number of loopback 3-tuples in a group. (In our implementation, $\alpha = 4$.) Our modified algorithm reduces the number of phase 2 iterations by a factor of α. Let $L_0, L_1, \ldots, L_{\alpha-1}$ be the loopback 3-tuples in a single group. The idea is to vary the number of connections made for each loopback 3-tuple in a group,

and have the tracking server differentiate between the loopback 3-tuples in the group by the magnitude of the difference between its two sampled measurements. Our algorithm makes $\beta \times 2^i$ connections to L_i ($\beta = 50$ in our implementation).

Denote by Δ_w the difference between the two source port measurements collected by the server for attacker 3-tuple w. With $\alpha = 1$, it is the same as described until now: L_0 shares the same table cell as an attacker 3-tuple w for which $\Delta_w = \beta + 1$. With $\alpha = 2$, there are two possibilities: either L_0 and L_1 each map to different attacker 3-tuples w_0 and w_1, or both collide with the same attacker 3-tuple w (i.e. $w = w_0 = w_1$). In the former case, the server will detect that $\Delta_{w_0} = \beta + 1$ and $\Delta_{w_1} = 2\beta + 1$. The difference for w_1 is larger since the number of connections attempted for L_1 is twice as much as L_0. In the latter case, where $w_0 = w_1 = w$, the server will detect that for a single w it has $\Delta_w = 1\beta + 2\beta + 1 = 3\beta + 1$. Thus, it concludes that the two loopback 3-tuples in the group must share the same table cell. This argument can be extended to higher values of α, see below.

Robustness against organic TCP connections. Organic TCP connections from the device spread uniformly across the T cells of the perturbation table and are thus unlikely to significantly affect a single cell in the short time our attack runs. Yet, such connections noise the server's measurements and we robustify our technique against such noise.

The phase 1 algorithm already handles noise: say $w \in S_i$ is a unique attacker 3-tuple and that some organic TCP connections were intertwined between the two source port measurements for w. If those TCP connections share the same table cell as w, then $\Delta_w > 1$. In this case, the algorithm determines that w is not unique. It does not compromise correctness: the algorithm continues iterating until covering all table cells.

In phase 2, we cannot rely on an algorithm that searches for precise differences so we add safety margins. Specifically, we segment the *differences space* to 2^α disjoint segments $I_0, I_1, \ldots, I_{2^\alpha - 1}$, where $I_k = [k\beta + 1, (k+1)\beta]$. Our algorithm then maps each attacker 3-tuple to a segment. The idea is that since the noise is typically small enough, it will cause the difference to be slightly above the "noiseless" expected value $k\beta + 1$, but still below $(k\beta + 1) + \beta$, i.e., $< (k+1)\beta + 1$. In other words, the difference will still belong to the segment $[k\beta + 1, (k+1)\beta]$. Thus, given a difference in segment $[k\beta + 1, (k+1)\beta]$, we determine that it is a result of $k\beta$ loopback connections, and from the binary representation of k we can reconstruct which L_i's were mapped to this w. These are precisely the L_i's in which 2^i appears as an addendum in the deconstruction of k into a sum of powers of two. To summarize, our phase 2 algorithm maps a loopback 3-tuple L_i to attacker 3-tuple w if and only if the i-th bit in the binary expansion of k_w (the segment number of w) is one.

4.5 Performance Analysis

4.5.1 Number of Iterations

We analyze the run-time (in terms of iterations) of phase 1 and of phase 2 in [10, §A]. For example, for $T = 256$ (the Linux case), phase 1 (Algorithm 2) needs 13.8 iterations on average to conclude. For $T = 256$ and a population of $N = 10^6$ devices, in order for the average ID collision count to be lower than one for the entire population, phase 2 (Algorithm 3 or Algorithm 4) needs 49.5 iterations on average to conclude. For a population of $N = 10^9$, phase 2 takes on average 60.4 iterations to conclude, and for $N = 10^{12}$, 69.9 iterations on average.

4.5.2 Dwell Time

In terms of dwell time (how long the browser needs to remain on the page for the process to complete), in phase 1 (Algorithm 2), each burst in a single iteration can be sent without waiting to the server's response. However, at the end of each iteration, the browser needs to wait for the server's response (the server updates the client on which ports should be used in the next iteration). Therefore, for phase 1, the required dwell time is the time needed for the browser to emit the 3 bursts, plus RTT, times the number of iterations. For phase 2 (Algorithm 4), the client and the server are not synchronized per iteration. Therefore, the dwell time is the time to emit the 2 bursts, times the number of iterations.

4.5.3 Device Bandwidth Use

In terms of packet count (and network byte count), iteration i of phase 1 emits $|S_i| + |S'_{i-1}| + |S_i|$ packets. Since $|S_i| = T - 1$ and $0 \leq S'_{i-1} \leq T - 1$, we have the packet count in each iteration between $2(T-1)$ and $3(T-1)$. Each iteration of phase 2 (Algorithm 4) consists of T packets (we do not count the loopback packets as they do not consume physical network resources). Each packet in a burst is a TCP SYN which has a minimal size (Linux TCP SYN is 60 bytes over IPv4 and 80 bytes over IPv6).

5 Implementation

Recent Linux kernels (versions 5.12-rc1 and above) use DHPS from RFC 6056 to generate TCP source ports. The implementation includes a few modifications to the RFC algorithm and parameter choice (§5.1) that require some adaptations of our technique (§5.2). We conclude this section by describing our implementation (§5.3).

5.1 Linux's DHPS Variant

Perturbation table. The Linux implementation has T=`TABLE_LENGTH`=256. The values of the perturbation table

are incremented by 2 (instead of 1), but this is immaterial to the attack. To simplify presentation, we ignore this detail for the rest of the discussion. Linux's implementation of CHECK-SUITABLEPORT verifies that the *port* is not locally reserved, and that the 4-tuple it forms, $(IP_{SRC}, port, IP_{DST}, PORT_{DST})$ is not already in use in the current network namespace (container). In Linux, $|K_2| = 128$.

Noise injection. A significant modification in the Linux implementation is that it randomly injects noise to the perturbation table when `__inet_hash_connect()` finds a suitable candidate in the first iteration of Algorithm 1. The relevant table cell is then incremented twice (instead of once) with probability $\frac{1}{16}$.

5.2 Adapted Phase 2 Algorithm

The phase 1 algorithm already handles noise caused by organic TCP connections, as described in § 4.4, so noise injected by Linux is handled identically. The adapted phase 2 algorithm should map the correct attacker 3-tuple for any given loopback 3-tuple, despite any noise injected by the kernel. We already describe in § 4.4 how the phase 2 algorithm can handle some noise (up to β additional table cell increments on top of the expected ones, per a table cell associated with an attacker 3-tuple w). Recall, we strive to use as high as possible an α value since it reduces the run-time of phase 2. Thus, we seek to make α as large as possible while keeping β higher than the noise induced by the device's Linux kernel and sporadic TCP connections.

The noise is maximal when w corresponds to a table cell which has a total of $(2^\alpha - 2)\beta$ connections. (The hardest task is to distinguish segment $[(2^\alpha-2)\beta+1, (2^\alpha-1)\beta]$ from $[(2^\alpha-1)\beta+1, 2^\alpha\beta]$.) The amount of noise for $(2^\alpha-2)\beta$ has a binomial distribution $\mathrm{Bin}((2^\alpha-2)\beta, \frac{1}{16})$. To support a population of 10^6 devices, we want an error probability of 10^{-6} to correctly find all the L_i values in a given device, which requires all $\frac{64}{\alpha}$ iterations of phase 2 to succeed. (Our implementation tests 64 loopbacks in phase 2, see § 6.4.) Therefore, we require $Prob(\mathrm{Bin}((2^\alpha-2)\beta, \frac{1}{16}) \geq \beta) \leq \frac{\alpha}{64}10^{-6}$. For $\alpha = 4$, this yields $\beta \geq 1244$ which is impractical since this implies $1244 \times (2^4 - 1) = 18\,660$ connections to be made for each group of loopback 3-tuples.

We now show how we can tweak the algorithm to support $\alpha = 4$ with a low β value. For this, we observe that there is exactly one w where $\Delta_w \geq 2^{\alpha-1}\beta$; it is exactly the w that shares the counter with $L_{\alpha-1}$. All other L_i's together cannot contribute more than $(2^{\alpha-1}-1)\beta$ to any counter. Thus, if we put the w whose $\Delta_w \geq 2^{\alpha-1}\beta$ aside, we are left with differences which are $\leq (2^{\alpha-1}-1)\beta$. This upper-bound forms the worst case, with the noise distribution $\mathrm{Bin}((2^{\alpha-1}-1)\beta, \frac{1}{16})$. Again we require $Prob(\mathrm{Bin}((2^{\alpha-1}-1)\beta, \frac{1}{16}) \geq \beta) \leq \frac{\alpha}{64}10^{-6}$. This time, for $\alpha = 4$, we get $\beta \geq 50$. Using $\beta = 50$ results in a manageable number of connections.

To summarize, we put aside the w whose $\Delta_w \geq 2^{\alpha-1}\beta$, we find L_i's that belong to all other w's and associate all the remaining L_i's (which were not associated to any other w) to the w whose $\Delta_w \geq 2^{\alpha-1}\beta$. This technique allows using $\alpha = 4$ with $\beta = 50$ to support 1 million devices. The value β grows slowly with the number of supported devices, e.g., supporting 1B devices requires $\beta = 73$.

5.3 Device Tracking Technique Prototype

We implemented a proof-of-concept of our device tracking technique. The client (snippet) is designed to run on both Chrome and Firefox and is implemented in approx. 300 lines of JavaScript code. We also implemented a client in Python that helped us during development and testing. The server is implemented in approx. 1000 lines of Go code. We describe key aspects of our implementation below and provide additional, more minor, implementation details in § 5.4.

The client and server exchange information via HTTP. In our implementation, the server acts as a "command-and-control" host: the client runs in a loop, requests the next command from the server, and executes it. This shifts much of the complexity from the client to the server and makes it easier to update the implementation without changing a complicated JavaScript implementation for multiple browsers.

Chrome and Firefox clients. Our client implementation for Google Chrome emits bursts of TCP connections by utilizing WebRTC. The client passes the list of destinations as a configuration for the `RTCPeerConnection` object and triggers the burst with the `setLocalDescription()` API. The advantage of WebRTC, compared to standard `XMLHttpRequest` or `fetch()` APIs, is that it allows to create connections at a rapid pace, which helps decreasing the overall dwell time. For Firefox, WebRTC is not applicable, since its implementation invokes `bind()` on the sockets before it calls `connect()` (see § 5.4.1). Instead, our Firefox implementation uses `XMLHttpRequest`.

Tracking server. The server uses `libpcap` to capture TCP SYN packets. It associates the TCP SYN packet to an active tracking client based on the source IP address and destination port. The server also stores the source port for later processing. For any attack-related incoming TCP SYN, our server replies with TCP RST (+ACK), except for the HTTP/HTTPS port on which the attacker's web server listens, of course. This way, upon receiving the RST, the client immediately discards the socket and does not send any further packets on it. This also has the advantage of keeping Linux's kernel connection table relatively vacant – otherwise, we risk hitting the process file descriptor limit (for Chrome, the limit is 8192).

Handling retransmissions. When a TCP SYN is left unanswered, Linux retransmits the SYN packet. This has the potential to confuse our server by having more source ports mea-

surements than expected. The Linux retransmission timeout is 1 second (`TCP_TIMEOUT_INIT` in `include/net/tcp.h`). Therefore, we might encounter this situation depending on the client's network and the RTT to the tracking server. To cope with retransmissions, the server deduplicates the TCP SYN packets it receives based on the combination of source IP, source port, and destination port fields.

5.4 Minor Implementation Details

5.4.1 Supporting Firefox

WebRTC vs. `XmlHttpRequest` Linux uses the revised DHPS for assigning TCP source ports when `connect()` is invoked with an unbound socket. This is the standard practice for establishing TCP connections from a client. However, it is also technically possible to establish a TCP connection from a client by first invoking `bind()` on a socket with a zero local port (instructing the kernel to pick a source port for the socket), and then applying `connect()` to it. When `bind()` assigns a source port number to the socket, the kernel has no information regarding the destination, and therefore it cannot use DHPS. Instead, Linux uses Algorithm 1 of RFC 6056 in this case. While not intended to be used by TCP clients, Firefox does in fact use `bind()` for its WebRTC connections. Therefore, in Firefox we resort to using `XmlHttpRequest`, which emits HTTP/HTTPS requests.

Re-connect attempt at the HTTP level When Firefox's HTTP connection attempt fails or times out, Firefox retries the connection at the HTTP level, i.e. it tries to establish a new TCP connection (as opposed to a kernel retry which uses the original connection parameters). These additional TCP connection attempts with their newly-generated TCP source ports cannot be easily distinguished from the expected TCP connection attempts. The phase 1 algorithm might be affected from re-connections when an attacker 3-tuple receives > 2 source port measurements. In such a case, the algorithm cannot analyze the attacker 3-tuple, and should discard it (in the present round), so that it is would not be flagged as "unique". The phase 2 algorithm was modified too: the client contacts the server after *each* group of loopback 3-tuples are tested (instead of after all groups). Further, for each burst of TCP connections the client makes, it also *waits* for each connection to complete (in contrast to the Chrome implementation). This prevents concurrent loopback connections and re-connections, which may corrupt source port measurements. Computing Δ_w is a bit more complicated, since as a result of re-connections, we might get > 2 source port measurements (more than expected). Consequently, we compute the difference between consecutive source ports measured, and set Δ_w to the maximum difference. We expect all consecutive differences to be 1 (up to some noise), except a single consecutive difference, which will correspond to the loopback increment we seek. This is because we do not allow re-connection attempts to overlap with the loopback connection attempts. For example, if we get 4 source port measurements s_0, s_1, s_2, s_3 then we set $\Delta_w = \max(s_1 - s_0, s_2 - s_1, s_3 - s_2)$. The algorithm continues normally from this point onward.

5.4.2 Private Network Access

The Private Network Access draft standard [22], which is implemented in Chrome, does not block our attack. If the snippet page is served over a secure context (HTTPS), then for XHR requests to loopback addresses, the browser will first attempt to send pre-flight HTTP OPTIONS requests to these destinations. These TCP connection attempts (SYN packets) to the loopback addresses suffice for our attack. Moreover, the Private network Access standard only applies to HTTP protocol traffic, and therefore WebRTC traffic to loopback addresses is not covered by it.

5.4.3 Scalability

In principle, the technique can scale very well. A technical limitation we need to consider is that we can only associate up to 65455 ports per (IP_{SRC}, IP_{DST}) address pair (the port range is 1-65535, with some 80 ports blocked by Chrome). This is not an issue for clients coming from different IP addresses. However, it may be a limitation when multiple clients behind one IP (NAT). We can address it by using more server IP addresses and load balance clients.

To maximize the gain from each server IP address, it should carefully manage the assignment and release of ports from a per-client-IP pool. In phase 1, each client consumes at most $2(T-1)$ destination ports during each iteration. In each iteration of phase 1, a batch of $T-1$ new destinations (ports) can be obtained from the server in real time (assigned from the pool). When the iteration is complete, the server determines which destinations are added to the client's list, and which are released back to the pool. Phase 1's worst case for ongoing consumption is $T-1$ ports (the client's expanding list). In phase 2, each client consumes exactly T destinations (the full client list, from phase 1). This list of destinations is allocated to the client for several seconds (the duration of phase 2). Thus, the peak consumption for each client is $2(T-1)$, so the algorithm can sustain $\frac{65455}{2(T-1)}$ simultaneous clients (with the same IP address) per server IP address. For $T = 256$, the server can sustain 128 clients (behind NAT) per server IP.

5.4.4 Handling packet drops

Packets may get dropped in a network due to congestion or routing changes. We adapt the tracking server to withstand moderate packet loss. First, we detect that there are too few packets for a specific attacker 3-tuple. Then, we find the burst for which the packet is missing by timing analysis: the largest time difference between captured packets is probably where a packet is missing. For the burst where the packet is missing,

we may still find the attacker 3-tuple for loopback address L_i by examining if there is a valid gap in TCP source ports of another attacker 3-tuple. If not, we rerun the test for this loopback address.

6 Evaluation

We use our proof-of-concept implementation to evaluate the device tracking technique against Linux-based devices. Our experiments answer the following questions:

1. Do we get a consistent device ID across browsers, tabs, and browser privacy modes?
2. Do we get a consistent device ID across networks, in both IPv4 and IPv6, across containers, and across VPNs?
3. Do we get a consistent device ID when the user browses other sites during the attack?
4. What is the dwell time required by our attack?
5. Is the attack applicable to Android devices?
6. How does the attack scale in terms of CPU and RAM? Is it suitable for large-scale tracking?

Setup. We deploy our tracking server in two Amazon EC2 regions: `eu-south-1` (IPv4 and IPv6) and `us-west-2` (IPv4 only). Each server is a `t3.small` instance, with 2 vCPU cores, 2GB of RAM and 5Gbps network link.

We tested three Ubuntu 20.04 Linux client devices: (i) HP Spectre x360 laptop (Intel Core i7-7500U CPU with 16GB of RAM) with kernel v5.13.19; (ii) ASUS UX393EA laptop (Intel Core i7-1165G7 CPU with 16GB RAM) with kernel v5.15.11; and (iii) Intel NUC7CJYH mini-PC (Intel Celeron J4005 CPU, 8GB RAM) with kernel v5.15.8.

6.1 Browsers

We demonstrate that we get a consistent device ID with our client snippet on Google Chrome (v96.0.4664.110) and Mozilla Firefox (v96.0) browsers (the latest versions at the time of writing). Since Chrome dominates the browser market, our optimizations are geared towards it.

We tested Chrome with our two tracking servers, both on IPv4 and IPv6. We verified that we get a consistent device ID across multiple tabs and browser modes, i.e., regular mode and incognito mode (one of the goals of this mode is to bolster privacy by thwarting online trackers). For Firefox, we verified that the modified tracking technique as depicted in § 5.4.1 works over the Internet and that the ID is identical to the one obtained via Chrome (cross-browser consistency).

6.2 Networks, NATs, VPNs and Containers

Our attack targets the client device, which operates in a variety of environments. It might access the Internet via a VPN, run the browser in a container or use a network behind a NAT. This section evaluates whether, and to what extent, these environments affect our attack.

Network	Port Rewriting?	Throttling?	IPv4 / IPv6
EduRoam	No	No	✓/ NA
University Guest	Yes	No	✗/ NA
Landline ISP 1	No	No	✓/ ✓
Landline ISP 2	No	No	✓/ NA
Landline ISP 3	No	No	✓/ NA
Landline ISP 4	No	IPv4 only	✓*/ ✓
Cable ISP 1	Yes	No	✗/ NA
Cellular ISP 1	IPv4 only	IPv4 only	✗/ ✓
Cellular ISP 2	IPv4 only	IPv4 only	✗/ ✓
Cellular ISP 3	No	Yes	✓*/ NA

* With slowed-down TCP SYN bursts.

Table 1: Tested networks

Networks and NATs. We tested our attack on multiple networks, on both landline and cellular ISPs, with IPv4 and IPv6. Table 1 summarizes our results. Our technique yielded a consistent ID for all tested networks which support IPv6. With IPv4, we found that some networks rewrite the TCP source port value (probably due to in-path port-rewriting NATs). Since our attack relies on observing the device-generated TCP source ports, it failed to obtain an ID on such networks. For IPv4 networks that do not rewrite TCP source ports, and for a given device, we got a consistent device ID, identical to the ID obtained from IPv6 networks for the same device (cross-network consistency, including cross IPv4/IPv6 consistency).

NATs are generally deployed on customer premise equipment (CPE) or at the ISP; the latter is often referred to as carrier-grade NAT (CGN). Importantly for our attack, many NAT implementations preserve the clients' TCP port selection. Mandalari et al. [17, Table I] showed that 78% of the tested NATs preserve TCP source ports. Their study covers over 280 ISPs that used 120 NAT vendors. Richter et al. [21] found that 92% of the CPE NATs they have identified in non-cellular networks preserve TCP source ports. For CGN deployments, Richter et al. found that on cellular networks, port preservation is less common: about 28% of the cellular networks with CGNs exhibit such a behavior. Among the non-cellular ISPs that use CGNs, 42% preserve ports. These measurements are in-line with our tested networks in Table 1.

To further assess the applicability of our attack, we deployed servers on four cloud providers (Azure, AWS, Google Cloud, and Digital Ocean) across 25 different regions and tested whether the network rewrites TCP source ports under IPv4. (IPv6 connections are less likely be NATed, as illustrated by our measurements in Table 1, since IPv6 addresses are abundant.) Our experiment shows that all tested regions and cloud providers do *not* rewrite TCP source ports. We list

the cloud providers and regions we tested in Appendix C.

Another issue we faced, mainly with cellular IPv4 ISPs, is traffic throttling (see Table 1). Such networks limit the packet rate of our TCP SYN bursts by dropping some packets. This may be due to traffic shaping or security reasons (SYN flood prevention). To address this problem, we spread our TCP SYN bursts over a longer time, thus increasing the overall dwell time of our technique.

VPNs. We tested our technique with a client device that is connected to an IPv4 VPN. We examined two popular VPN providers: TunnelBear VPN and ExpressVPN. In both cases, we set up a system-wide VPN configuration using Ubuntu's Network Manager. We note that the vast majority of VPNs (in particular, the two VPNs we tested) do not support IPv6 [25].

We tested two locations with TunnelBear VPN, Germany and Finland, and found that the TCP source ports are preserved. However, TunnelBear's exit nodes throttle the outbound VPN traffic, so we expect the attack to work with slowed-down bursts. For ExpressVPN, our attacks succeeded on 7 out of 10 exit nodes tested in North America and Western Europe. The failed attempts were due to TCP port rewriting. The device ID we obtained through VPNs was identical to the one obtained when the device was using a regular network, and the dwell time we experienced with ExpressVPN was comparable to a regular network with the same RTT. We conclude that, in many cases, VPN exit nodes do not rewrite TCP source ports, which allows our technique to work. This demonstrates that VPNs do not inherently protect against our technique.

Linux containers. We deployed two docker containers on the same host and ran our Python client implementation on each. Both runs produced the same device ID, identical to the host device ID, as expected (cross-container consistency). We conducted the experiment with `containerd` version 1.4.12, `runc` version 1.0.2 and `docker-init` version 0.19.0.

6.3 Active Devices

In this experiment, we demonstrate that a consistent device ID is obtained when the client simultaneously visits other websites during the attack. To this end, we opened multiple tabs on Chrome during the attack and visited sites that are listed under Alexa's Top 10 Sites. On each test, we arbitrarily chose 3 to 4 websites from the list (this includes "resource-heavy" sites such as Yahoo, YouTube, and QQ). In all of our tests, we verified that we get a consistent device ID, concluding that our technique successfully withstands organic TCP connections generated by the victim device during the attack.

6.4 Dwell Time

The dwell time is the execution time of our attack, the sum of phase 1 and 2 completion times. Phase 1 completion time is affected by the number of iterations it takes to collect T unique attacker 3-tuples. The measured average is 15 iterations, which is expected to be slightly higher than the theoretical average of 13.8 iterations we computed in [10, §A] due to the Linux injected noise (see § 5.1). The network round trip time (RTT) to the tracking server also affects phase 1's completion time since, at the end of each iteration, the client contacts the server to determine which of the attacker 3-tuples tested in this iteration are unique. Phase 2's completion time is mainly affected by the number of loopback groups tested. In our implementation, we tested a fixed number of 64 loopback groups. The RTT has significantly less impact on phase 2 since we do not wait for server responses.

We measured the dwell time for our Chrome client when using $\alpha = 4$ and $\beta = 50$ (see § 4.4) against our two tracking servers. With an average RTT of 50ms, we measured an average dwell time of 7.4 seconds. With an average RTT of 275ms, we measured an average dwell time of 13.1 seconds. Overall, our results for Chrome show a dwell time of 5-15 seconds (10 seconds on average).

For Firefox, the dwell time is on the order of several minutes, even under lab conditions, because of in-browser throttling. For this reason, our implementation uses more balanced values for α, β: $\alpha = 2$ and $\beta = 10$. Lowering the β value for Firefox has an advantage since the number of connections to the loopback interface (on phase 2) decreases, hence making the attack run faster under throttling. We have not attempted optimizing our Firefox implementation further.

6.5 Android

At the time of writing, there is no Android device that uses DHPS (see the discussion on § 3). To verify that the attack is applicable to Android, we manually introduced the DHPS code into the 5.4 kernel of a Samsung Galaxy S21 device. (Linux's DHPS implementation is conveniently located in one file, making this change self-contained.)

During our experiments with the Samsung device, we observed a `netfilter` rule that limits the rate of incoming new TCP SYN packets to an average of 50 per second. This rule limits our attack in phase 2 when many connections to the loopback interface are attempted (an outgoing TCP SYN packet in the loopback interface eventually becomes an incoming TCP SYN packet, which is subject to this rule). To work around this restriction, we modified the α, β parameters of the attack to $\alpha = 2$ and $\beta = 10$. This results in $10 + 2 \times 10 = 30$ loopback connections being attempted for each loopback group, which is below the limit imposed by this rule. With this configuration, our lab experiments show that the attack on Chrome yields a consistent device ID when the device switches networks, with a dwell time of 18-21 seconds.

We presume that the offending rule (causing us to adjust the attack) is not general to Android but rather, it is Samsung-

specific since we did not find evidence of it in the Android Open Source Project (AOSP) code.

6.6 Resource Consumption

Server side CPU utilization. The tracking server needs very little CPU resources. In both phases, the server-side calculation simply involves going over the iteration data and finding the port pairs in which the difference is above some threshold.

Server side RAM consumption. While computing an ID for a device, the tracking server needs to keep (i) a list of allocated destination ports; and (ii) the most recently observed source port per each destination port. In the worst case, the list can be up to $2T$ pairs of ports (with $T = 256$, the maximum list length is 512). Each port number is 16 bits, so each port pair takes 4 bytes, and overall the server needs up to $8T = 2$KB per actively tested device. Suppose the device dwell time is W seconds, and the rate of devices per second incoming funnel is R, then at a given moment, there will be $R \cdot W$ tested devices, which requires $8T \cdot R \cdot W$ bytes. For Linux-based devices using Chrome, we have $T = 256, W = 10s$, so the tracking server RAM consumption is $R \cdot 20$KB per device. For example, if $R = 10^6$ devices/sec, the server needs 19.07GB RAM. Keep in mind that R does not represent the total number of devices the server needs to support, it is only the rate at which the server is required to measure devices (which is much lower than the total number of supported devices).

Client side CPU utilization. In our experiments, CPU utilization by our tracking logic was negligible.

Client side RAM consumption. Most of the client-side RAM consumption imposed by our tracking snippet is due to attempting to create TCP connections. In the attack, the tracking server responds with TCP RST, prompting the victim's browser and kernel to release this memory quickly. In an experiment, we measured an overhead of at most 30MB in client memory (tested on Chrome), which is sufficiently low to allow the attack even on low-end devices.

7 Countermeasures

The root cause of our attack is the ability to detect hash collisions in G_{K_2} via a shared perturbation table cells. The ideal solution would be to create a private perturbation table instance per network namespace and interface. By doing so, we prevent the attacker's ability to detect "stable" hash collisions (on the loopback interface). The problem with this solution is that its memory consumption could be high when many containers are spawned, or many interfaces are present.

To mitigate the attack when the perturbation table is shared across interfaces, DHPS must ensure that either hash collisions are much less frequent or that detecting collisions is much more difficult. In line with the above, we propose below modifications to DHPS and summarize how they were applied to the Linux kernel in a recent patch to mitigate our attack. We also analyze in Appendix B an alternative algorithm proposed by RFC 6056, "Random-Increments Port Selection Algorithm" (Algorithm 5, [13, Section 3.3.5]), showing that the trade-off it offers cannot simultaneously meet the functionality and security goals from [13, Section 3.1] (see § 3).

Increase the table size T. This makes hash collisions much less frequent. For example, instead of $T = 256$ we can use $T = 256K = 262,144$. This consumes 0.5MB of RAM (assuming each table entry is 16 bits). The attack now takes $\times 1024$ time due to the need to cover all T table cells. (In the patch issued for Linux, T was increased to $64K$.)

Periodic re-keying. Changing the secret key in DHPS results in a different table index being accessed and a different port offset from which candidate enumeration will begin. After re-keying, any table collision information previously obtained by an attacker becomes useless. The trade-off is that a source port that was previously chosen for the same 3-tuple could be chosen again. This might prevent a TCP connection from being established (see § 3). To reduce the chance for connectivity issues, re-keying should not be performed too frequently. (In the patch issued for Linux, re-keying is performed every 10 seconds, balancing functionality and security.)

Introduce more noise. The Linux kernel team also increased the noise in perturbation table cell increments to make detecting collisions more difficult. Now, each increment is an integer chosen uniformly at random between 1 and 8.

7.1 Network Security Measures

Besides mitigating our attack at its core (the DHPS algorithm), network security appliances such as firewalls, IDS, or IPS could thwart it to some degree. Since our attack requires a non-negligible number of connection attempts to the same set of attacker destinations, a network security appliance could detect this condition and limit future connection attempts to those destinations. This would require the attacker to slow the rate of connection attempts, presumably to the point where the attack becomes impractical. Crucially, however, by doing so, the security appliance might flag legitimate traffic as malicious: for example, when multiple users behind NAT enter a resource-heavy website in a short time span.

A better mitigation strategy would be to use *on-host* logic, which can be part of a *host* IPS (HIPS) solution, a personal firewall, or in-browser security logic. The on-host logic can detect the internal loopback traffic that is generated as a result of our attack and rate limit TCP connection attempts to *closed* ports on the loopback interface. We do not expect standard applications to exhibit similar behavior, making this countermeasure effective and with a low false-positive rate.

8 Conclusion

This paper illustrates a flaw in the DHPS algorithm for selecting TCP source ports – an algorithm which was proposed by RFC 6056 and recently adopted by Linux. We exploit this algorithm to track Linux devices across networks, browsers, browser privacy modes, containers, and VPNs. The key observation of this paper is that the attacker can detect *collisions* in the output of a keyed hash function used in DHPS via sampling, in an attacker-prescribed manner, TCP source ports generated from the global kernel state induced by DHPS. By indirectly observing such collisions for *loopback* 3-tuples inputs, the attacker calculates an invariant device ID (agnostic to the network the device is connected to) since it only relies on the secret hashing key which is generated at system startup. Interestingly, this result does not rely on the choice of the hash function, because the hash function's output space is small enough for collisions to happen naturally even in a very moderate number of TCP connections. We implement the attack for Linux devices and Chrome and Firefox browsers, and demonstrate it across the Internet in practical, real-life scenarios. We propose changes to the DHPS algorithm that mitigate the root cause of our technique. Lastly, we worked with the Linux kernel team to integrate countermeasures into the Linux kernel, which resulted in a recent security patch.

9 Vendor Status

We reported our findings to the Linux kernel security team on February 1st, 2022. In response, the Linux team developed a security patch which was incorporated in versions 5.17.9 (and above) and 5.15.41 (the LTS versions that include DHPS). The Linux kernel issue is tracked as CVE-2022-32296.

10 Availability

Our code is publicly available online, along with instructions for reproducing our results, in https://github.com/0xkol/rfc6056-device-tracker.

Acknowledgements

We thank the anonymous Usenix 2022 reviewers for their thoughtful comments, suggestions and feedback. This work was supported in part by the Hebrew University cyber security research center. Yossi Gilad was partially supported by the Alon fellowship.

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: dusting the web for fingerprinters. In *ACM CCS '13*, 2013.

[2] Jonathan Berger, Amit Klein, and Benny Pinkas. Flaw Label: Exploiting IPv6 Flow Label. In *IEEE SP*, 2020.

[3] Christian Dullweber. Expire favicons on cache deletion. https://chromium.googlesource.com/chromium/src/+/c1ce99f4ed59, March 2021.

[4] David Dworken and Eric Dumazet. tcp: change source port randomizarion [sic] at connect() time. https://github.com/torvalds/linux/commit/190cc82489f46f9d88e73c81a47e14f80a791e1a, February 2021.

[5] Vinay Goel. An updated timeline for privacy sandbox milestones. https://blog.google/products/chrome/updated-timeline-privacy-sandbox-milestones/, June 2021.

[6] Sabrina Jiang. Fingerprinting With Atomic Counters in Upcoming Web Graphics Compute APIs, 2020.

[7] Amit Klein. Cross Layer Attacks and How to Use Them (for DNS Cache Poisoning, Device Tracking and More). In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[8] Amit Klein and Benny Pinkas. DNS Cache-Based User Tracking. In *NDSS*, 2019.

[9] Amit Klein and Benny Pinkas. From IP ID to Device ID and KASLR Bypass. In *USENIX Security*, 2019.

[10] Moshe Kol, Amit Klein, and Yossi Gilad. Device Tracking via Linux's New TCP Source Port Selection Algorithm (Extended Version). https://arxiv.org/pdf/2209.12993.pdf, 2022.

[11] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. DRAWN APART : A Device Identification Technique based on Remote GPU Fingerprinting. In *NDSS*, 2022.

[12] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2), April 2020.

[13] M. Larsen and F. Gont. Recommendations for Transport-Protocol Port Randomization. RFC 6056.

[14] Martin Lastovicka, Stanislav Spacek, Petr Velan, and Pavel Celeda. Using TLS Fingerprints for OS Identification in Encrypted Traffic. In *NOMS*, 2020.

[15] Chun-Han Lin, Shan-Hsin Lee, Hsiu-Chuan Huang, Chi-Wei Wang, Chia-Wei Hsu, and ShiuhPyng Shieh. DT-Track: Using DNS-Timing Side Channel for Mobile User Tracking. In *DSC*, 2019.

[16] Linux. /include/net/tcp.h. https://github.com/torvalds/linux/blob/master/include/net/tcp.h.

[17] Anna Maria Mandalari, Miguel Angel Diaz Bautista, Francisco Valera, and Marcelo Bagnulo. NATwatcher: Profiling NATs in the Wild. *IEEE Communications Magazine*, 2017.

[18] Microsoft. Settings that can be modified to improve network performance. https://docs.microsoft.com/en-us/biztalk/technical-guides/settings-that-can-be-modified-to-improve-network-performance, June 2021.

[19] Android Open Source Project. Android Common Kernels – Feature and launch kernels. https://source.android.com/devices/architecture/kernel/android-common, May 2022.

[20] A. Ramaiah, R. Stewart, and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961.

[21] Philipp Richter, Florian Wohlfart, Narseo Vallina-Rodriguez, Mark Allman, Randy Bush, Anja Feldmann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. A Multi-Perspective Analysis of Carrier-Grade NAT Deployment. In *IMC*, 2016.

[22] Titouan Rigoudy and Mike West. Private network access. https://wicg.github.io/private-network-access/, June 2021.

[23] Konstantinos Solomos. Security: Favicon cache can reveal entries through a leaky side channel. https://bugs.chromium.org/p/chromium/issues/detail?id=1096660, June 2020.

[24] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Tales of Favicons and Caches: Persistent Tracking in Modern Browsers. In *NDSS*, 2021.

[25] Sven Taylor. Best VPNs with FULL IPv6 Support in 2022. https://restoreprivacy.com/vpn/best/ipv6/, May 2022.

[26] John Wilander. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/, March 2020.

[27] Henrik Wramner. Tracking Users on the World Wide Web. http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/wramner_henrik_11041.pdf, 2011.

Appendices

A Another Use Case: Traffic Measurement

It is possible to count how many outbound TCP connections are established by a device in a time period using the above techniques. This is useful for remote traffic and load analysis, e.g. to compare the popularity of services. For example, it is possible to remotely measure the rate at which outbound TCP connections are opened by a forward HTTP proxy. This can be used to estimate how many concurrent clients the HTTP forward proxy serves. And quoting [4]: "In the context of the web, [counting] how many TCP connections a user's computer is establishing over time [...] allows a website to count exactly how many subresources a third party website loaded. [...] Distinguishing between different users behind a VPN based on distinct source port ranges, [...] Tracking users over time across multiple networks, [...] Covert communication channels between different browsers/browser profiles running on the same computer, [... and] Tracking what applications are running on a computer based on the pattern of how fast source ports are getting incremented".

This attack builds on the phase 1 technique (§ 4.2). To mount the attack, the attacker needs to have client access to the forward proxy. For simplicity we assume that the attacker is simply one of the proxy's clients. The attacker first runs the phase 1 logic with the client side being a standalone script/software (not inside a browser) that runs on a machine that has client access to the forward proxy, and establishes T attacker 3-tuples that conform to the T perturbation table counters. This needs to be done only once, and ideally when the target device is relatively idle.

Next, the attacker can poll the TCP source ports p_i for each of these T attacker 3-tuples at time t. The attacker then polls the TCP source ports p'_i at time $t' > t$. Denote by ρ the total number of ephemeral ports in the system: $\rho = max\_ephemeral - min\_ephemeral + 1$, and suppose no counter advanced more than $\rho - 1$ steps (keep in mind that the attacker's first poll also increments each counter by 1), then the total number of TCP connections established by the device between t and t' is:

$$\sum_{i=0}^{T-1} ((p'_i - p_i - 1) \mod \rho)$$

This measurement can be repeated as long as the device does not restart.

B Analysis of RFC 6056's Algorithm 5

RFC 6056's Algorithm 5 increments a global counter by a random value between 1 and N, where N is configurable. In order to avoid connection id reuse, RFC 6056's Algorithm 5 should ensure that the counter does not wrap

around in less than $2 \cdot MSL$ seconds, where MSL is the *server* TCP stack parameter. The original TCP RFC 793 sets $MSL = 120$. The default value for Windows servers (the registry value `TcpTimedWaitDelay`) is 60 seconds [18]. In Linux, $MSL = 30$ seconds (evident from the kernel constant `TCP_TIMEWAIT_LEN`$= 2 \cdot MSL = 60$ [16]). The average progress per TCP port in RFC 6056's Algorithm 5 is $\frac{N+1}{2}$, therefore in order not to wrap around before $2 \cdot MSL$ seconds have elapsed, the following condition is necessary (but not sufficient):

$$2 \cdot MSL \cdot \frac{N+1}{2} \cdot r < R$$

Where R is the port range (for Linux/Android, $R = 60999 - 32768 + 1 = 28232$), and r is the outbound TCP connection rate. The above upper bound for N is not tight, because it assumes that each connection is short lived, i.e. terminated very shortly after it is established. If a connection is long lived, then its TIME_WAIT phase is achieved after even more ports are consumed, thus lowering the bound for N. In an anecdotal test with a Linux laptop running Ubuntu 20.04.3 and Chrome 96.0.4664.110, we opened several tabs for media-rich websites and got 737 TCP connections in a 64.5 seconds time interval, thus we measured $r = 11.4$ connections/s. For Linux servers, this yields $N \leq 81$, and for Windows servers, this yields $N \leq 19$. As we noted above, this is a very loose upper bound for N. And it is quite possible that r higher than 11.4 is common in some scenarios. But even setting $N = 81$ yields low security since it reduces the entropy of the TCP source port by 8.5 bits (in the Linux server case), from $\log_2 28232 = 14.8$ to $\log_2 81 = 6.3$ (10.6 entropy bit reduction, to 4.2 bits in the Windows case). This seems unacceptable security-wise, and so Algorithm 5 fails to deliver a practical trade-off between security and functionality.

Interestingly, RFC 6056 suggests $N = 500$ without explanation how this value is obtained.

C Cloud Providers Experiment

In this experiment, we deployed servers on multiple cloud providers across different regions and tested whether their networks rewrite TCP source ports. Our test uses a utility that contacts a reference server on a bound source port (random but known to the utility), to which the server replies with the observed source port. Our experiment shows that all 25 tested regions across 4 cloud providers (Azure, AWS, Google Cloud and Digital Ocean) do *not* rewrite TCP source port. Table 2 summarizes the results.

Cloud provider	Region	Preserve Ports?
Azure	Australia East	✓
	Sweden Central	✓
	Central US	✓
	Central India	✓
	UK West	✓
	Korea Central	✓
AWS	Oregon	✓
	Milan	✓
	Canada	✓
	Sau Paulo	✓
	Singapore	✓
	Cape Town	✓
	Hong Kong	✓
Google Cloud	Las Vegas	✓
	Santiago	✓
	Madrid	✓
	Taiwan	✓
	Jakarta	✓
	Melbourne	✓
	Zurich	✓
Digital Ocean	Amsterdam	✓
	Bangalore	✓
	Singapore	✓
	New York	✓
	Toronto	✓

Table 2: Tested cloud providers and regions under IPv4