You Can Rand but You Can't Hide: A Holistic Security Analysis of Google Fuchsia's (and gVisor's) Network Stack

Inon Kaplan
Independent researcher

Ron Even
Independent researcher

Amit Klein
Hebrew University of Jerusalem

***Abstract*—This research is the first holistic analysis of the algorithmic security of the Google Fuchsia/gVisor network stack. Google Fuchsia is a new operating system developed by Google in a "clean slate" fashion. It is conjectured to eventually replace Android as an operating system for smartphones, tablets, and IoT devices. Fuchsia is already running in millions of Google Nest Hub consumer products. Google gVisor is an application kernel used by Google's App Engine, Cloud Functions, Cloud ML Engine, Cloud Run, and Google Kubernetes Engine (GKE). Google Fuchsia uses the gVisor network stack code for its TCP/IP implementation.**

We report multiple vulnerabilities in the algorithms used by Fuchsia/gVisor to populate network protocol header fields, specifically the TCP initial sequence number, TCP timestamp, TCP and UDP source ports, and IPv4/IPv6 fragment ID fields. In our holistic analysis, we show how a combination of multiple attacks results in the exposure of a PRNG seed and a hashing key used to generate the above fields. This enables an attacker to predict future values of the fields, which facilitates several network attacks. Our work focuses on web-based device tracking based on the stability and relative uniqueness of the PRNG seed and the hashing key. We demonstrate our device tracking techniques over the Internet with browsers running on multiple Fuchsia devices, in multiple browser modes (regular/privacy), and over multiple networks (including IPv4 vs. IPv6). Our tests verify that device tracking for Fuchsia is practical and yields a reliable device ID.

We conclude with recommendations on mitigating the attacks and their root causes. We reported our findings to Google, which issued CVEs and patches for the security vulnerabilities we disclosed.

I. Introduction

In this paper, we review the security of the network protocol stack used by Google Fuchsia operating system [1], based on the network stack of Google gVisor kernel [2]. Our primary use case is Fuchsia device tracking, but our findings apply to other use cases and facilitate network attacks against Fuchsia and gVisor-based applications.

The main scenario is a Fuchsia device running a browser, which navigates to a website. This website serves a web page containing an HTML+JavaScript tracking snippet. The snippet works in coordination with the website's server backend, and the server computes a device ID based on information extracted from the TCP/IP protocol headers of the device's traffic to the server. The same device ID is calculated for all websites that employ this tracking technique, and thus, the device can be tracked across its navigation through websites, time, and networks ("cross-site tracking"). To elaborate, two or more websites may collude in order to track a web user, by individually calculating the device ID when the user's device visits each website, using it to identify the same user as it traverses the colluding websites, even though the user may log-in to the colluding websites under different accounts, from different networks and from different browsers. It is important to note that in general, without a common device ID (be it based on protocol header fields, hardware fingerprints or other identifiers), it would be impossible for colluding websites to track visitors across sites, because each website may know its users by different, site-specific names (e.g. their account names/numbers).

We introduce several tracking techniques based on vulnerabilities in the Fuchsia/gVisor implementation of various TCP/IP protocol header fields. Mainly, we exploit the small seed space in the PRNG used by the network stack, the PRNG predictability, the weakness in the hashing function used to calculate some TCP protocol fields, the small hash key space in the IPv4 ID generation algorithm, and hash collisions in the IPv4 ID table. These result in weaknesses in how the TCP Initial Sequence Numbers (ISN), TCP timestamps, and TCP source ports are generated, which we exploit to recover the network stack's PRNG seed. This seed is generated at the operating system startup and remains in effect until the operating system shuts down and, therefore, can be used as a stable device ID. An alternative, computation intensive technique uses the UDP source ports to extract this seed. A third technique exploits the IPv4 ID field generation algorithm to extract the hashing key used by it, which, similarly to the seed, can be used as another stable device ID.

We demonstrated our device tracking techniques on various Fuchsia devices, including a Google Nest Hub Max device (smart home speaker+display), a Google Pixelbook Go laptop, an Intel NUC device (mini-PC), and two virtual devices (running on QEMU). We tested them through 7 networks (2

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240122
www.ndss-symposium.org

cellular, 2 VDSL, 2 Fiber, and one cable) via WiFi hotspots and Ethernet connections. We tested IPv4 and IPv6 connectivity and browser regular mode vs. privacy mode (Chrome's "Incognito" mode) against our proof-of-concept (PoC) servers in two continents. We measured success rates, dwell time required, and compute time and found that all were realistic for tracking at scale – we had complete success in all 63 tests. The dwell time needed for the PRNG seed extraction was an average of seven milliseconds over IPv4 and 116 milliseconds for the hash key extraction. Their compute time was three and five seconds, respectively. Over IPv6, however, the dwell time was only two milliseconds on average, and the compute time was only one millisecond.

Our TCP-based techniques require the tracking website to observe the TCP timestamp and TCP sequence number as the Fuchsia device generates them. Likewise, the IPv4-based techniques require direct observation of the IPv4 ID field. This means that a device behind a forward proxy (and particularly Tor) is not vulnerable to these techniques. However, at this time, to the best of our knowledge, the Fuchsia browser does not support Tor (or a forward proxy in general). Moreover, a forward proxy does not affect the UDP-based technique since our Javascript code observes the UDP source ports locally.

The same techniques and underlying weaknesses can be used to mount other attacks. Since we expose the PRNG seed, we can reproduce its outputs. Therefore, our techniques can predict network protocol fields generated by the PRNG (at least the part determined by the PRNG output). These fields are the TCP source port, the TCP ISN, the TCP timestamp, and the UDP source port. Attacks that involve IPv4 ID predictability can also be mounted based on obtaining the IPv4 ID hashing key. In addition, our attacks expose the device's internal (private) IPv4 address, even when the device is behind a NAT.

Our holistic approach was key in developing practical tracking techniques (and other attacks). Each individual vulnerability we uncovered has a limited (albeit non-negligible) impact. But the ability to take in the interplay between the vulnerabilities and combine them made the more potent device tracking attacks possible.

A. *Fuchsia and gVisor*

Fuchsia is an open-source operating system developed by Google. They describe it as "a general purpose operating system designed to power a diverse ecosystem of hardware and software" [1]. It has a clean-slate design, including a new microkernel called "Zircon". Fuchsia components, subsystems, drivers, etc., are written mostly in Rust, except Zircon which is written mostly in C++ (with some C and Assembly) and Fuchsia's GUI which is written in Dart using the Flutter SDK. Fuchsia employs modern software engineering concepts such as a capability model, software packages, software isolation, sandboxing, namespaces, modularity and updateability. Fuchsia first appeared in an open source repository in 2016 [3], and it was first deployed (to 1st generation Google Nest Hub devices) in 2021.

It is widely speculated that Fuchsia is slated to replace Android on smartphones and tablets [4], [5], [6], with some indications that Samsung and Huawei plan to release Fuchsia-based smartphones in the foreseeable future [7], [8], though some sources are more reserved about the timelines [9]. Google has already started to update shipped products (Google Nest Hub 2nd Gen, Google Nest Hub Max) to Fuchsia [10], [11]. Since in Q3 2021 alone "Google's Nest Hub ... [had] 1.5 million units shipped" [12], we estimate that **many millions of Google Nest Hub devices run Fuchsia at present**.

As such, Fuchsia's security, particularly its network stack and vulnerability to tracking, is of great importance, especially when looking several years ahead. Moreover, we (the security research community) are uniquely positioned time-wise to look at a future operating system and address its security issues before it becomes ubiquitous, i.e., before such time that patching it yields major production and deployment challenges.

Fuchsia's network stack runs in user-space (part of Fuchsia's "Connectivity" core service), and is built upon Google gVisor's network stack. According to the Google gVisor GitHub page, "gVisor is an application kernel, written in Go [...] gVisor is an application kernel for containers. It limits the host kernel surface accessible to the application while still giving the application access to all the features it expects" [13]. According to [14], gVisor is used in Google's App Engine, Cloud Functions, Cloud ML Engine, Cloud Run, and Google Kubernetes Engine (GKE). Since researching Fuchsia's network stack equates to researching gVisor, some of our research may also apply to non-Fuchsia gVisor use cases.

It should be noted that the Fuchsia source code tree also contains an alternative, purpose-built network stack ("NetStack3"). Since it is not the default stack used by Fuchsia, we did not invest efforts into analyzing it. However, per our source code analysis, it appears that it is also vulnerable to a device tracking attack, which we describe in App. A.

B. *Network Protocol Attacks*

1) Device Tracking: Device tracking is a fundamental threat to Internet users' privacy. Used for personalization of advertisement and surveillance, it, by design, undermines the anonymity of users as they navigate across sites ("cross-site tracking"), networks, ISPs, and browser modes (regular vs. privacy mode).

Until recently, trackers used 3rd-party cookies as a tracking mechanism due to their convenience and prevalence (cookies are a web standard supported by all browsers). However, nowadays, major browser vendors severely limit the scope and usability of 3rd-party cookies [15], [16], rendering them almost useless for practical cross-site tracking. Trackers are, therefore, looking for alternative web-based tracking techniques, specifically ones that can reliably track devices in today's privacy-aware browser environment. For example, browser vendors nowadays use separate caching contexts for sites, frames, etc., thus rendering many shared-cache attacks useless. Likewise, browser privacy modes pose a challenge for trackers since browser vendors attempt to separate them from the main browsing mode completely.

Over the years, multiple tracking techniques were developed. Some were based on detecting immutable properties of the device's operating system, kernel, and/or hardware to produce a *fingerprint* of the device, ideally (but hardly always) unique among all potentially trackable devices. Other techniques attempt to *tag* a device by placing a unique marker in the device, e.g., forcing the device to cache one or multiple resources [17, Section IV]. Both approaches are not without problems. One fingerprinting challenge is to come up with a sufficient number of information bits to tell devices apart. Another fingerprinting challenge is the "golden image" scenario, wherein an organization deploys thousands of identical hardware and software devices, which makes it extremely difficult to tell devices apart. A significant challenge for the tagging approach is the continuous efforts of browser vendors to prevent information sharing[1] across browsing contexts and especially across the privacy mode gap.

Thus, finding a web-based tracking technique to overcome the golden image and isolated browsing context scenarios is even more challenging. In this paper, we describe techniques based on extracting immutable kernel data (PRNG seed, hashing key) that can be used as a device ID that remains intact from system startup to system shutdown (typically weeks or months). Because our device ID is a value of a global kernel object, it is agnostic to the website in which it is calculated, the browsing context (1st-party or 3rd-party, privacy mode), the network to which the device is connected to (including IPv4 vs. IPv6, NAT, and source IP address variability), the location of the tracking server and its IP addresses, and the time of ID measurement. Our techniques yield a stable device ID of 31 bits (PRNG seed) / 32 bits (hashing key) / 63 bits (their combination).

2) Additional Attacks: While our main focus in the paper is device tracking, our attacks retrieve the Fuchsia/gVisor PRNG seed and IPv4 ID hashing key, which are used to calculate numerous network protocol fields. As such, our results can be used to predict TCP ISN, TCP and UDP source ports, IP IDs and PRNG outputs and to disclose internal IPv4 addresses, which can facilitate diverse network attacks. We did not attempt to mount any of the non-tracking attacks.

C. Our Contribution

Our contribution is four-fold:

- We provide the first holistic security analysis of the Fuchsia (and gVisor) TCP/IP network protocol stack, focusing on how "high entropy" protocol header fields are generated. Specifically we review how Fuchsia (and gVisor) generates the TCP initial sequence number, TCP timestamp and TCP source port, UDP source port, IPv4/IPv6 ID and IPv6 flow label fields. We report vulnerabilities in all these fields except for the IPv6 flow label.[2]
- Since we holistically analyze all the protocol fields collectively, we observe that the vulnerabilities we found can be chained and exploited in concert to mount multiple variations of device tracking attacks. Our attacks also disclose the PRNG seed, the IPv4 ID hashing key and the device's internal (private) IP address.
- We conduct experiments and report their results, which demonstrate these vulnerabilities over networks and connections on various Fuchsia devices. We set up two PoC servers on two continents to show that our tracking techniques work across the Internet and are agnostic to the PoC server location and IP addresses.
- We provide recommendations for addressing and mitigating our attacks.

II. Related Work

A. Fuchsia/gVisor Network Stack

Operating systems and kernels often have proprietary implementations of their network protocol stacks. As such, each network stack implementation needs to be assessed individually. Network stacks of well-established kernels such as Microsoft Windows, Linux (also the kernel of Android), and Apple XNU (the kernel of macOS, iOS, and all other Apple products) are well-studied. For example, the algorithms used to populate protocol header fields are scrutinized in [18], [19], [20], [21]. Yet the gVisor network stack has not attracted attention in this context, with the scarcely existing research focusing on finding generic code vulnerabilities across the kernel [22]. We submit that our work is the first systematic analysis of algorithmic weaknesses in how gVisor populates network protocol header fields.

The gVisor network stack uses Go's built-in pseudo-random number generator (PRNG). In [23], it is shown that when the attacker obtains 607 consecutive full PRNG outputs (63 bits), the attacker can reconstruct all previous PRNG outputs and predict all the succeeding outputs. However, this condition does not hold with any of the gVisor PRNG use cases we reviewed. Particularly, UDP source ports are generated using the most significant 31 bits of the PRNG output, taken modulo 49536. This is incompatible with the attack in [23].

B. Holistic Security Analysis of Network Stack Algorithms

Many research works focus on the algorithmic security of a single protocol header e.g. IPv4 ID [18], IPv6 flow label [24] and TCP ISN [25], [26]. A cross-layer attack is described in [20] where a (flawed) PRNG is broken by sampling IPv6 flawlabel and used to predict the next UDP source port, but this stems from a single vulnerability (predictable PRNG) that happens to have multiple manifestations. Finally, [27], [28], [29] review entire network stacks, but focus on non-algorithmic vulnerabilities (e.g. buffer overflows).

[1]In general, tagging also faces a severe challenge in multiple-browser environments, but this is irrelevant to Fuchsia at the time of writing.

[2]Fuchsia/gVisor always set the IPv6 flow label to 0, thus its effective entropy is 0.

Our work shows that by taking a holistic view, we can combine seemingly unrelated "minor" vulnerabilities into powerful attacks. For example, to find the IPv4 ID hashing key, we combine four independent vulnerabilities in four protocol fields: TCP ISN, TCP timestamp and TCP source port (to find the internal IP address), and finally IPv4 ID itself.

C. Fuchsia/gVisor IPv4 ID Scheme

Unlike other network protocol fields, the gVisor algorithm for generating IPv4 IDs (and IPv6 fragment IDs) is a degenerate version of the Linux 3.16-5.1 algorithm for generating IPv4 IDs. The original Linux IPv4 ID algorithm was analyzed and exploited in the context of Linux attacks, e.g., in [18]. The Linux algorithm involves increments of random quantities depending on the duration between bucket accesses, whereas the Fuchsia/gVisor algorithm degrades this to simple increments (++). This allows a somewhat different attack from [18] against gVisor, which is faster than [18] and requires fewer packets. For example, in [18], the attack required a dwell time of several seconds due to the need to send two batches of 400 packets several seconds apart. In contrast, our attack requires a single batch of 250 packets, necessitating an average dwell time of only 116 milliseconds. This can be critical for tracking scenarios that can guarantee only a sub-second dwell time, e.g. an interstitial page or ad rotation.

In addition, back in 2019, extracting the source IPv4 address using WebRTC was possible [30], but this leak has since been fixed [31]. We overcame this challenge in two alternative ways, as explained in § VI-A and § VI-B.

D. Device Tracking and Other Attacks

An extensive review of state of the art in web-based device tracking is provided in [32], [18], [24], [20], [21]. Specifically, [18], [24], [20], [21] describe network protocol header -based device tracking attacks against Linux, Android and Windows. None of the network protocol attacks reviewed applies as-is to Fuchsia/gVisor since gVisor has its proprietary implementation of the TCP/IP network stack (up to the special case of IPv4 ID explained above). In addition, Fuchsia incorporates a relatively recent Google Chrome browser (v111 at the time of writing), which eliminates many browser-based weaknesses that enabled older device tracking attacks.

There are still a few attacks that the above resources do not cover. Several browser-based fingerprinting techniques are described in [33]. These can detect the browser engine, the OS, the browser mode (regular/privacy), browser extensions, and the instruction set architecture (ISA). For Fuchsia, this yields very few bits of information since there is only one browser (Chrome), no extensions, and only two relevant ISAs (x64 and ARM). A row hammer-based device fingerprinting is described in [34]. They mention that it "takes almost 3 minutes to extract a fingerprint," which renders this technique impractical for many device tracking use cases. Recently, [35, Table VI] described several cross-site tracking techniques. Specifically for Chrome, the disclosed new techniques are either in experimental features (Private State Token API, FLEDGE API), subsequently fixed (favicon cache, Alt-Svc), or of limited impact (CORS Preflight – only effective for two hours). A device tracking technique based on the Widevine EME DRM standard is described in [36]. While they did not test browsers on Fuchsia, they do note that Chrome for Android (probably the closest target to Fuchsia they tested) is not vulnerable to their attack, which suggests that Chrome on Fuchsia is not vulnerable as well.

III. Notations, Definitions, Basic Observations, Attacker Model

A. Notations and Conventions

- Throughout this paper, unless stated otherwise, we implicitly assume that addition and subtraction are carried modulo 2^{32}.
- We denote by $MSB(x)$ the most significant byte of x (typically x is a 32-bit quantity).
- We designate the time since boot (in time units of $[u]$) when a TCP field X is generated as $t^X_{[u]}$.

B. Definitions

Fuchsia uses Jenkins' "One-at-a-Time Hash" function [37] to generate values of different protocol fields, such as the TCP timestamp (TS), the TCP Initial Sequence Number (ISN), and the TCP source port. This function can be expressed as $Sum32(HashBytes(B, V))$ where B is an array of bytes and V is a 32-bit hashing key, with the result hash value of 32 bits. Alg. 1 is a pseudo-code for $HashBytes$, and Alg. 2 is a pseudo-code for $Sum32$.

Algorithm 1: HashBytes

```
Input: An array of Bytes $B$ and a value $V$
Output: A 32-bit integer (hash of $B$ and $V$)
begin
    foreach $b$ in $B$ do
        $V \leftarrow V + b$;
        $V \leftarrow V + (V \ll 10)$;
        $V \leftarrow V \oplus (V \gg 6)$;
    end
    return $V$
end
```

Algorithm 2: Sum32

```
Input: A value $V$
Output: A 32-bit integer
begin
    $V \leftarrow V + (V \ll 3)$;
    $V \leftarrow V \oplus (V \gg 11)$;
    $V \leftarrow V + (V \ll 15)$;
    return $V$
end
```

C. Observations

- The function $x \longmapsto x + (x \ll n)$ is easily invertible. It can be written as $x \longmapsto (2^n + 1)x \mod 2^{32}$, thus inverting it amounts to multiplying by $(1-2^n)(1+2^{2n}+2^{4n}+\cdots)$ which takes very few register-only instructions.
- The function $x \longmapsto x \oplus (x \gg n)$ is easily invertible using very few register-only instructions [38].
- It follows from the above that $Sum32(\cdot)$ is easily invertible, and so is $HashBytes(B, \cdot)$ (when B is known).
- If $HashBytes([b], v)$ is known (where $[b]$ is a one-cell array containing the byte b), then it is easy to calculate $b + v$ (follows from the above and the definition of $HashBytes$).

D. Attacker Model

In our attack model, the attacker (website) embeds a JavaScript snippet inside an HTML page served to the client device. On the device, the page is consumed by a browser; thus, the attacker's Javascript code runs on the browser in the context of the attacker's website. This Javascript code instructs the browser to communicate with the attacker's web server, e.g., using WebRTC, so that the server has some protocol data from the device at the end of the interaction, allowing it to calculate a device ID for it (the PRNG seed and/or the $hashIV$ hashing key). Ideally, the dwell time on the attacker's page and the amount of network traffic generated by the snippet should be minimal.

On the server side, the attacker sets up an Internet-connected machine with enough RAM (90GiB suffices to support all the attacks described in § IV-B and § IV-C). Ideally, the machine would also have sufficient computing power (dozens of cores) to calculate a device ID in real time. For the TCP attacks (§ IV-B and § IV-C), the machine needs to have two Internet IP addresses: IP_1 and IP_2. For the IPv4 ID attacks (§ VI), the PoC server has 250 IPv4 addresses.

A cross-site tracking attack involves two or more **colluding** websites. Each colluding website calculates the device IDs of their visitor devices, ideally in real-time. Our device ID can be calculated using a **website-specific infrastructure** (IP addresses and domains names), independent of other colluding websites, and thus cannot be trivially flagged as a common tracking infrastructure.

IV. Obtaining the PRNG Seed

A. The Network Stack PRNG

Fuchsia's network stack uses an instance of the built-in Go PRNG with a 31-bit seed[3]. The network stack PRNG instance is seeded during the network stack initialization (at system startup) using random bytes obtained from a cryptographically strong source, and is never reseeded since. The PRNG then advances deterministically. During the network stack initialization, the PRNG is used to generate three 32-bit "secrets" in deterministically prescribed PRNG invocation offsets: S_{ISN} – the TCP ISN hashing key, S_{PS} – the TCP source port hashing key and S_{TS} – the TCP timestamp hashing key. While Fuchsia is running, the PRNG instance is used to generate UDP source ports.

[3]Nominally the seed initialization function `rng.Seed()` takes a 64-bit integer argument, but this argument is used $\mod 2^{31}-1$, and moreover if the result is 0, it is changed to 89482311. Thus there are only $2^{31}-2$ effective seeds: $[1,\ldots,2^{31}-2]$. See also [39].

B. Obtaining the PRNG Seed with Just Four TCP/IPv4 SYN Packets

1) Overview: The attack consists of three "phases." First, the attacker extracts a value denoted as J_{TS}. Formally defined later, it is an intermediate calculation of a TCP packet's timestamp value. Next, using J_{TS}, the attacker extracts a value denoted as J_{ISN}, an intermediate value in calculating a TCP circuit's ISN value. The last phase consists of extracting the seed itself using J_{TS} and J_{ISN}. As a byproduct of the attack, the attacker also obtains the device's internal (private) IP address. Note that this attack does not assume anything about how the PRNG is advanced, except that it is a known, deterministic algorithm.

2) Setup: The attacker server has two Internet addresses, IP_1 and IP_2. In offline, the attacker computes two multi-maps – Q and W (32GiB each), as described below. These multi-maps are derived from the values of IP_1 and IP_2. The snippet initiates two TCP connections (SYN packets) to IP_1 and IP_2 (four TCP connection attempts altogether). To generate the connection attempts **rapidly**, the snippet uses WebRTC TURN. For ease of reference, we denote these connections $SYN_{IP_1,1}, SYN_{IP_1,2}, SYN_{IP_2,1}, SYN_{IP_2,2}$.

3) Extracting J_{TS} Candidates: Alg. 3 describes the TCP timestamp (TS) calculation for a TCP connection and defines J_{TS}.

```
Algorithm 3: TS Timestamp Calculation
Input: The TCP connection addresses $IP_{SRC}, IP_{DST}$
       in big-endian format, a 32-bit integer secret
       $S_{TS}$
Output: A 32-bit integer (TCP timestamp)
begin
    $J_{TS} \leftarrow HashBytes(IP_{SRC}, S_{TS})$;
    $offset \leftarrow HashBytes(IP_{DST}, J_{TS})$;
    return $Sum32(offset) + t^{TS}_{[ms]}$;
end
```

In this phase of the attack, the attacker produces a small set of J_{TS} candidates (20 in expectation).

Offline, the attacker computes a multi-map Q:

$$Sum32(HashBytes(IP_2, J_{TS})) - Sum32(HashBytes(IP_1, J_{TS})) \longmapsto J_{TS}$$

by going over all 2^{32} possible J_{TS} values.

The online attack phase starts when the Fuchsia device browses the tracking HTML page. The attacker collects the packets $SYN_{IP_1,2}$ and $SYN_{IP_2,1}$ and extracts their TS values. The two packets are created δ milliseconds apart. Therefore:

$$TS_{SYN_{IP_1,2}} = Sum32(HashBytes(IP_1, J_{TS})) + t$$
$$TS_{SYN_{IP_2,1}} = Sum32(HashBytes(IP_2, J_{TS})) + t + \delta$$

Subtracting, we get:

$$TS_{SYN_{IP_2,1}} - TS_{SYN_{IP_1,2}} - \delta = Sum32(HashBytes(IP_2, J_{TS})) - Sum32(HashBytes(IP_1, J_{TS}))$$

Since the packets $SYN_{IP_1,2}$ and $SYN_{IP_2,1}$ are generated in rapid succession, δ is small. Our experiments log $\delta < 11$ (milliseconds) at maximum, so we can safely assume $\delta < 20$. The attacker iterates over $0 \le \delta < 20$, and for each such δ the attacker collects all precomputed J_{TS} candidates (all $Q(TS_{SYN_{IP_2,1}} - TS_{SYN_{IP_1,2}} - \delta)$ values).

This attack results in 20 J_{TS} values in expectation, which the attacker forwards to the next phase. At the end of the next phase, the attack converges on the single correct J_{TS}.

4) Extracting J_{ISN}: Alg. 4 describes the TCP Initial Sequence Number (ISN) calculation for a TCP connection and defines J_{ISN}.

Algorithm 4: TCP ISN Generation

```
Input: The TCP 4-tuple
       $IP_{SRC} : sport, IP_{DST} : dport$, and the ISN
       secret $S_{ISN}$. Addresses are big-endian, ports
       are little-endian.
Output: A 32-bit integer (ISN)
begin
    $J_{ISN} \leftarrow HashBytes(IP_{SRC}, S_{ISN})$;
    $v_1 \leftarrow HashBytes(IP_{DST}, J_{ISN})$;
    $v_2 \leftarrow HashBytes(sport, v_1)$;
    $v_3 \leftarrow HashBytes(dport, v_2)$;
    return $Sum32(v_3) + t^{ISN}_{[64ns]}$;
end
```

Extracting J_{ISN} is not straightforward. Unlike J_{TS}, J_{ISN}'s calculation involves the TCP source and destination ports, and in a NAT use case (which is very common in IPv4 networks), the TCP source port used by the host is not exposed to the attacker. Additionally, it uses a fine-grained clock (64 nanosecond resolution) to offset the hash value, which is more challenging than J_{TS}'s millisecond clock.

From the previous step, the attacker has several J_{TS} candidates. The attacker needs to iterate through all of them and carry out the procedure described below, but for simplicity, we assume a single J_{TS} from now on. Denote by $t^{TS}_{[ms]}$ the time (in milliseconds) since boot when the TS field of a TCP packet was generated. Since the attacker already extracted J_{TS}, the attacker can calculate it easily:

$$t^{TS}_{[ms]} = TS - Sum32(HashBytes(IP_{DST}, J_{TS}))$$

Each pair of packets sent by the browser, $SYN_{IP_i,1}$, $SYN_{IP_i,2}$ for $i \in \{1, 2\}$, yields a set of J_{ISN} candidates (through the to-be-presented calculation). The attacker calculates the intersection of the two sets, which, with a very high probability, yields a single J_{ISN} value (the correct J_{ISN}). Since for incorrect J_{TS} the intersection is very likely to be empty, this also reveals the correct J_{TS}.

Let $t^{ISN}_{[64ns]}$ be defined similarly to $t^{TS}_{[ms]}$ as the time (in 64 nanosecond resolution) since boot when the ISN field was generated. We now describe how to generate a set of J_{ISN} candidates for a single pair of TCP SYN packets (sent to the same attacker IP address). The attacker starts by enumerating over the possible $t^{ISN}_{[64ns]}$ values for the **first** SYN packet. The attacker does this based on the $t^{TS}_{[ms]}$ the attacker has for this packet. Since the ISN field for a TCP SYN packet is generated a few microseconds before the TS field, and since the $t^{TS}_{[ms]}$ value is rounded down to the nearest millisecond, we have:

$$15625 \cdot t^{TS}_{[ms]} - 1563 \le t^{ISN}_{[64ns]} < 15625 \cdot t^{TS}_{[ms]} + 15625$$

($15625 = {}^{1000000}/_{64}$ is the conversion rate between clock ticks, and $1563 = {}^{100000}/_{64}$ represents 100 microseconds in 64 nanosecond clock ticks, a very coarse upper limit for the delay between ISN and TS generation time). Thus the attacker has $15625 + 1563 = 17188$ candidates for $t^{ISN}_{[64ns]}$ and hence for $Sum32(v_3) = ISN - t^{ISN}_{[64ns]}$.

Since $Sum32(\cdot)$ is invertible, and so is $HashBytes(bytes, \cdot)$ given known $bytes$ (e.g., $dport$ in this case), the attacker can generate 17188 candidates for v_2. Going backward further in $HashBytes$ is not trivial since the attacker typically does not know $sport$ (due to NAT).

As a background for the next step, we describe how Fuchsia generates ephemeral TCP source ports. Fuchsia employs "Algorithm 3" of RFC 6056 [40] to select TCP source ports. In detail, Fuchsia has a variable P ("port hint") incremented for every new outbound TCP connection. For a would-be TCP connection to $IP_{DST} : dport$, Fuchsia assigns the source port

$$(Sum32(HashBytes(IP_{SRC}||IP_{DST}||dport, S_{PS})) + P) \bmod 49536 + 16000$$

Where S_{PS} is the 32-bit TCP source port secret (this assumes that the 4-tuple is presently unused at the device, which in our case always holds). Fuchsia generates source ports in the range of $[16000, 56635]$.

We split the source port of the first packet $sport$ to the high and low bytes i.e. $sport = 256 \cdot s_H + s_L$ where $62 \le s_H \le 255$ and $0 \le s_L \le 255$, and we observe that $v_2 = HashBytes(s_H, HashBytes(s_L, v_1))$. Now the attacker obtains $HashBytes(s_L, v_1)$ by enumerating over s_H (194 values). Employing the observation in § III-C, the attacker obtains $v_1 + s_L$, which we denote as $x_{t,s_H} = v_1 + s_L$ (the attacker has 194×17188 candidates for x_{t,s_H}).

For simplicity, we now assume that no organic outbound TCP connections are attempted by the system while our TCP measurements are taken. This means that P is only incremented due to our measurements. This is a valid assumptions since (as we show in Table II) the entire TCP measurement takes a few milliseconds). Moreover, we can easily mitigate organic connections by adding tolerance, but this complicates the discussion.

We use another observation based on the above: consecutive TCP source port allocations for the same destination IP address and port (given a fixed source address) are incremental. Up to a wrap-around (rare), we can assume that for the two consecutive TCP SYN packets to the same attacker destination, the source port of the second packet is the source port of the first packet plus 1.

We now look at the second packet and denote its fields and corresponding notations with a circumflex. Thus $\widehat{sport} = 256 \cdot s_H + (s_L + 1)$ (assuming $s_L < 255$). From this, the attacker has $\hat{v}_2 = HashBytes(s_H, HashBytes(0, x_{t,s_H} + 1))$, and the attacker can roll forward $\hat{v}_3 = HashBytes(dport, \hat{v}_2)$. Finally, the attacker calculates $\hat{t}^{ISN}_{[64ns]} = \widehat{ISN} - Sum32(\hat{v}_3)$, and eliminates false positives by verifying that:

$$15625 \cdot \hat{t}^{TS}_{[ms]} - 1563 \leq \hat{t}^{ISN}_{[64ns]} < 15625 \cdot \hat{t}^{TS}_{[ms]} + 15625$$

Since the probability of a random candidate to fulfill the above is 17188×2^{-32}, there will be $(17188 \times 194) \times 17188 \times 2^{-32} = 13.34$ surviving candidates for x_{t,s_H}, in expectation.

The attacker extends each surviving x_{t,s_H} candidate into a J_{ISN} candidate by enumerating over all 256 possible s_L values, calculating $v_1 = x_{t,s_H} - s_L$, and going backward in $HashBytes$ (with the known IP_{DST} address) to get J_{ISN}. Thus, each x_{t,s_H} results in 256 J_{ISN} candidates, and so for a pair of SYN packets to the same destination, the attacker gets $256 \times 13.34 = 3416$ J_{ISN} candidates, in expectation. Note that with each J_{ISN} candidate, the attacker also has a corresponding candidate for the source port of the first packet ($sport = 256 \cdot s_H + s_L$) and the source port of the second packet ($sport + 1$).

Covering the cases wherein s_L or s_H wrap around is not difficult but left out due to space considerations.

The attacker applies the same procedure for the second pair of TCP SYN packets (that arrive at the second attacker IP address) to obtain another set of 3416 J_{ISN} candidates, in expectation. The attacker calculates the intersection of the two sets. When J_{TS} is correct, this yields a single value (the correct J_{ISN}) with very high likelihood because the probability for a match between random candidates is 2^{-32}. When J_{TS} is incorrect, the intersection is very likely to be empty. Thus, the attacker can determine the correct J_{ISN}. The attacker also obtained the source ports of all four packets as a by-product.

5) Extracting the PRNG Seed: Recall that S_{TS}, S_{ISN} and S_{PS} are generated from the PRNG at fixed offsets, making it easy to generate them given a seed. In offline, the attacker calculates a multi-map W by going over all $2^{31} - 2$ possible seeds, and for each seed S, add the mapping $S_{ISN} - S_{TS} \longmapsto [S, S_{TS}]$ to W.[4] Note that there are (expected) $\frac{2^{31}-2}{2^{32}} \approx 0.5$ elements in a W cell.

[4]Theoretically we need to keep S only, because S_{TS} is derived from it, but because deriving S_{TS} can take microseconds, and this needs to be done 2^{24} times, we trade-off memory to save time and keep S_{TS} in memory.

Denote the true seed as S^* and the corresponding secrets as $S^*{}_{TS}$ and $S^*{}_{ISN}$. Given J_{ISN} and J_{TS}, we use the following method of obtaining S^*. Recall that $J_{ISN} = HashBytes(IP_{SRC}, S^*{}_{ISN})$. The attacker goes 3 bytes backward in $HashBytes$ by enumerating all 2^{24} options of the least significant 3 bytes of IP_{SRC}. Partially inverting $HashBytes$ one more time, the attacker extracts $S_{ISN} + MSB(IP_{SRC})$. The attacker similarly computes $S_{TS} + MSB(IP_{SRC})$ from J_{TS}. Denote the latter expression by x.

The attacker then calculates

$$\begin{aligned}\Delta = (S_{ISN} + MSB(IP_{SRC})) - (S_{TS} + MSB(IP_{SRC})) \\ = S_{ISN} - S_{TS}\end{aligned}$$

Finally, $W(\Delta)$ is the list of candidate seeds (and their corresponding S_{TS}), and note that $(S^*, S^*_{TS}) \in W(\Delta)$ for the correct guess of the 3 least significant bytes of IP_{SRC}. Now, for a seed candidate S obtained from $W(\Delta)$ the attacker uses S_{TS} to calculate $x - S_{TS}$. For the correct seed, $x - S_{TS} = MSB(IP_{SRC})$. Since $MSB(IP_{SRC}) \leq 255$, the attacker can remove false positives by checking that $x - S_{TS} \leq 255$.

This ends up in very few seed candidates (one in expectation). With each seed candidate, the attacker also has a corresponding IP_{SRC} candidate. To filter the list further and find the correct seed, the attacker goes over the remaining seed candidates and, for each such seed candidate S, calculates its port secret S_{PS}. With S_{PS} and IP_{SRC} obtained, and since the source ports are computed as a by-product of the attack (part of the J_{ISN} extraction), the attacker can extract the packets' port hint (modulo 49536):

$$\begin{aligned}P_{SYN_{IP_i,j}} = sport - 16000 - \\ Sum32(HashBytes(IP_{SRC}||IP_i||dport, S_{PS})) \\ \mod 49536\end{aligned}$$

P is incremented by one after each source port assignment, regardless of the connection specifics. Assuming that $SYN_{IP_1,2}$ and $SYN_{IP_2,1}$ are sent in rapid succession, the attacker can postulate that in between these two packets, no other outbound TCP connection is attempted or at least only a few are (say, less than 10). The attacker can, therefore, filter seed false positive by verifying that $1 \leq P_{SYN_{IP_2,1}} - P_{SYN_{IP_1,2}} \leq 10$ (in modular arithmetic). This concludes the extraction of S^*.

C. Obtaining the PRNG Seed with Just Two TCP/IPv6 SYN Packets

Extracting the seed from IPv6 traffic is more straightforward, as NAT is rarely used for IPv6, and therefore the attacker can observe the client's source IPv6 address and source TCP/UDP port. Similarly to the IPv4 attack, the IPv6 attack requires an attacker server with two (IPv6) addresses. Unlike the IPv4 attack, the IPv6 attack only requires a single SYN packet per IP address. Let TS_1 and TS_2 denote their respective timestamps, and similarly for ISN_1 and ISN_2.

The attacker obtains J_{TS} candidates using the method from § IV-B, step 2. This requires an additional 32GiB multi-map Q', calculated in offline. The attacker then calculates the respective S_{TS} candidate per each J_{TS} candidate via inverting the $HashBytes$ using the (known, in the IPv6 case) source address IP_{SRC}.

In offline, the attacker computes a 24GiB multi-map W' by going over all possible seeds, and for every seed S, computing the timestamp secret S_{TS} and adding the mapping $S_{TS} \to S$ to W'. Given W', the attacker can generate a (small) set of seed candidates for each S_{TS} candidate. The expected list size (false positives) is ≈ 0.5 for every S_{TS} candidate.

To filter for S^* the attacker goes over all seed candidates, and for each candidate S, the attacker calculates the timestamp offset for the first packet, using its source address IP_{SRC} and its destination address IP_1. The difference between this offset and TS_1 is the TCP TS generation time in milliseconds since boot on the device. This is similarly calculated for the second packet. Denote these times by $t^{TS_1}_{[ms]}$ and $t^{TS_2}_{[ms]}$, respectively. Then, the attacker calculates the ISN offset for the two packets using the IP_{SRC}, and their respective destination addresses (IP_1, IP_2) and source and destination ports. By subtracting the result from ISN_1 and ISN_2 the attacker obtains the least significant 32 bits of the time since boot (in 64 nanosecond clock ticks) in which the respective sequence number was generated on the Fuchsia device. Converting $t^{TS_1}_{[ms]}$ and $t^{TS_2}_{[ms]}$ to 64ns clock tick units (i.e., multiplying by 15625) yields an approximation of the time (in 64 nanosecond ticks) the TCP TS was generated on the device, which is (up to) a few microseconds later than the ISN generation time. Thus, the attacker can compare this quantity per packet to the ISN generation time and filter out seed candidates whose derived ISN and TS generation times are too far from each other (modulo 2^{32}).

D. Obtaining the PRNG Seed with 0 UDP Packets

When a Fuchsia application sends a UDP datagram from an unspecified source port, Fuchsia assigns the datagram to a UDP source port using the PRNG. Since the source port pool is $[16000, 65535]$, this yields $log_2 49536 = 15.59$ bits of information gleaned from every source port. An attacker can obtain a series of consecutively assigned UDP source ports by instantiating WebRTC STUN objects without sending a single UDP datagram. This can be done, for example, by setting the WebRTC destination to 0.0.0.0, an invalid IPv4 destination address; WebRTC will bind a local UDP port, but the send operation will fail at the OS level, producing no packet. The allocated UDP source port number can be obtained by Javascript code; thus, an attacker Javascript code can obtain a sequence of UDP source ports (derived from PRNG outputs) without triggering additional traffic.

A long enough series can be used to extract the PRNG seed. Naïvely, one can enumerate over all $2^{31}-2$ possible seeds and roll each of them forward up to K steps to find a match with the UDP source ports obtained. K is the product of the average PRNG consumption (per day) and the maximum duration (since Fuchsia boot) the attacker wishes to "cover" with the technique (for example, 100 days). In § VII, we measure the average consumption of PRNG outputs to be 10,000-30,000 invocations/day. We can set $K = 10^7$ as a safe upper bound for 100 days. This results in $N \approx 2^{31}K \approx 2^{54.3}$ calculations. To be able to find the seed, the attacker needs $n = \lceil \frac{\log_2 N}{15.59} \rceil = 4$ consecutive UDP port samples. The CPU load in this approach is prohibitive (proportional to $N = 2^{54.3}$).

An alternative approach is to use a time-memory trade-off technique such as rainbow tables [41]. Formally, the attacker defines a function f from the Cartesian product of the seed space and the PRNG steps since boot space to the space of all possible sequences of n consecutive ports $f : [1, 2^{31} - 2] \times [0, K-1] \to [1, 2^{31} - 2] \times [0, K-1]$, where $f(s, r)$ is the sequence of n UDP source ports generated from seed s after rolling it forward r times, "cast" into the smaller space of size $|[1, 2^{31} - 2] \times [0, K-1]|$ using any suitable and fast hash function $g : [16000, 65535]^n \to [1, 2^{31} - 2] \times [0, K-1]$. The problem is thus to "invert" f for a particular value – the hash g of the sequence of n UDP source ports obtained in real-time. The inverse value corresponds (with sufficiently high probability) to the seed and number of PRNG invocations before the first port in the sequence.

In the rainbow table approach, we first choose a desired point on the time-memory trade-off curve $TM^2 = N^2$ [42] where T is the number of f evaluations and M is the number of basic memory units needed. As explained in App. B, each evaluation of f "costs" $607n$ basic operations, therefore we choose $T = 2^{28.7}$ (total $2^{39.9}$ basic operations), which yields $M = 2^{39.9}$. There is a one-time heavy offline calculation involving N evaluations of f and preparation of a RAM table whose size is M "memory units". This is quite demanding from the computation time perspective, and as such, we could not implement this approach in our PoC, but it is feasible for a budgeted attacker. This PRNG seed extraction technique is orthogonal to the previous attacks based on TCP and requires 0 network packets and minimal dwelling time. The downside of this technique is the CPU compute time (and, to a lesser degree, the RAM consumption) on the server side.

Note that we can get the success probability of the attack as close to 1 as needed by using multiple rainbow tables (increasing time and memory linearly).

V. Predicting UDP Ports

In this attack, we demonstrate that the Go PRNG used in Fuchsia and gVisor is weak, regardless of its effective seed size. That is, we assume nothing about the seed size and seeding procedure, and we only rely on the weakness of the PRNG advancement procedure. We exploit this weakness to show that UDP source ports are generated predictably.

When a UDP client starts a new UDP session without explicitly requesting a specific source port (a standard usage pattern), Fuchsia assigns it a source port using the PRNG:

$$port = (\texttt{rng.Int63()} \gg 32) \mod 49536 + 16000$$

Go's built-in PRNG uses ALFG (Additive Lagged Fibonacci Generator [43]) and is implemented in the `rng` class of the `rand` package. The PRNG uses an internal state, $R_0, \ldots, R_{rngLen-1}$, consisting of $rngLen = 607$ unsigned

63-bit integer values[5] and returns a pseudo-random 63-bit integer value `rng.Int63()`, described in Alg. 5. Our analysis is based on the particular way the PRNG is advanced, namely that $R_n = R_{n-607} + R_{n-273} \mod 2^{63}$. Define P'_n as the port number generated at step n of the PRNG minus 16000. Then $P'_n = (R_n \gg 32) \mod 49536$.

We begin with a generic observation about expressions involving shifts and sums. There are two possible independent reasons that for $a, b \in \mathbb{Z}$, $((a+b) \mod 2^{63}) \gg 32 \neq (a \gg 32) + (b \gg 32)$ (the right-hand side addition is carried out over $\mathbb{Z}$, i.e., non-modular arithmetic):

1) $(a+b)$'s calculation involves a carry from the 32 least significant bits of the sum — this means that the carry is "lost" by the right shift on a and b individually, therefore $(a+b) \gg 32 = (a \gg 32) + (b \gg 32) + 1$.
2) $a + b \geq 2^{63}$ – This means that $(a+b) \mod 2^{63} = a + b - 2^{63}$. When shifted, it results in $((a+b) \mod 2^{63}) \gg 32 = (a+b) \gg 32 - 2^{31} = (a \gg 32) + (b \gg 32) - 2^{31}$.

Now, substituting $a = R_{n-607}, b = R_{n-273}$, and applying $\mod 49536$ on both sides, there are four possible cases:

- $P'_n = (P'_{n-607} + P'_{n-273}) \mod 49536$
- $P'_n = (P'_{n-607} + P'_{n-273} + 1) \mod 49536$ (by reason 1)
- $P'_n = (P'_{n-607} + P'_{n-273}) - 2^{31} \mod 49536$ (by reason 2)
- $P'_n = (P'_{n-607} + P'_{n-273}) + 1 - 2^{31} \mod 49536$ (by both reasons).

Algorithm 5: Go's PRNG functions

```
$rngLen \leftarrow 607$;
$rngTap \leftarrow 273$;
$feed \leftarrow rngLen - rngTap$;
**Function** rng.Int63():
    $rngTap \leftarrow (rngTap - 1) \mod rngLen$;
    $feed \leftarrow (feed - 1) \mod rngLen$ ;
    $R_{feed} \leftarrow R_{feed} + R_{rngTap} \mod 2^{63}$;
    **return** $R_{feed}$;
**return**
```

Thus, by observing the previous 607 UDP source ports (generated by consecutive invocations of the PRNG), it is possible to predict (up to four values) the next UDP source port that will be assigned by Fuchsia ($P'_n + 16000$).

Similar to § IV-D, the attacker's Javascript obtains a series of 607 consecutive UDP source ports without triggering additional traffic. It is well known that Chrome has a 500 WebRTC connection limit [44]. However, when a Fuchsia device has an active dual-stack (i.e., both IPv4 and IPv6 addresses are allocated to it – a very common scenario), it is sufficient to instantiate only 304 WebRTC STUN objects since Chrome will allocate two source ports per each WebRTC STUN object (one for IPv4 and one for IPv6). Both ports are accessible to the Javascript code, and thus the Javascript code has access to 608 consecutive UDP source ports.

If no other UDP connection is established during the assignment of the WebRTC STUN ports, the above equations hold and can be used to predict (up to four values) the next UDP source port Fuchsia will assign.

VI. Obtaining $hashIV$ (IPv4)

The 16-bit IP ID field in the IPv4 packet header is generated in the Fuchsia kernel using a hash function H with a 32-bit secret hashing key denoted as $hashIV$ generated randomly at boot time. Fuchsia uses Jenkin's lookup3 hash [45] truncated to 11 bits as H, but our attacks are oblivious to the choice of H, as long as the hashing key size is small (32 bits). The hash calculation over the packet network addresses and the transport protocol number $H(IP_{SRC}, IP_{DST}, transport, hashIV)$ is used as an index to a table of 2048 buckets – initialized to random values at boot time. The bucket value is used (modulo 2^{16}) as the IP ID, and then the bucket is incremented by one. It is, therefore, likely that IP IDs obtained almost simultaneously, which are observed to have a difference of 1, have been taken from the same bucket.

A. A Straightforward Attack

Using the above observation and employing the hash collision attack concept from Klein et al. [18], the attacker can extract $hashIV$. The attacker sets up an Internet server with $L = 250$ public IPv4 addresses attached to it. First, the attacker mounts the TCP attack on IPv4 traffic IV-B, to obtain the client's source IP address. Next, the embedded Javascript code instructs the client's browser to send a UDP packet to each of the L IPv4 addresses, e.g., using WebRTC TURN. Denote this list of IPs as R. The attacker collects the IP IDs of each packet arriving at its server, along with the packet destination address. Once all L packets are collected, the attacker finds possible hash collisions: pairs of packets identified by their destination IP addresses (IP_x, IP_y) whose IP IDs have a difference of 1. This likely means they are taken from the same bucket, though it is still possible that this happens randomly due to two different buckets having values with a difference of exactly one. With the set of possible collisions C, the attacker enumerates the 2^{32} possible $hashIV$ candidates, and for each candidate h, the attacker calculates the number of pairs in C which collide under h, i.e. the size of the set:

$$\{(IP_x, IP_y) \in C | \\ H(IP_{SRC}, IP_x, 17, h) = H(IP_{SRC}, IP_y, 17, h)\}$$

(Recall that the UDP protocol transport number is 17).
Theoretically, the correct h would yield a complete match. However, considering that C may contain noise (as explained above), a more relaxed strategy can be used so the attacker can select the h with the maximal match count.

In App. C, we show that the choice of $L = 250$ yields failure probability $< 10^{-6}$, which suffices to support reliable tracking of 1 million devices. Of course, this does not guarantee that

[5]In reality, the integers are 64 bits, but it can be easily seen that the most significant bit can be ignored for `rng.Int63()`.

two devices will not be assigned the same device ID, i.e., that two devices will have the same seed; in fact, there are $\binom{1000000}{2}/2^{32} \approx 116$ pairs (in expectation) of such devices in an ensemble of 1,000,000 devices. This is an inherent drawback of the technique; however, the $hashIV$ can be combined with the PRNG seed to form a 63-bit device ID which is likely to be unique for 1,000,000 devices.

In our PoC, we relaxed the matching criteria so that the match only considers pairs in C, i.e., we do not discard keys with pairs outside C. This yields a simpler and faster matching algorithm at the price of increasing false positives to some extent (insignificant for the PoC).

B. An Independent (Self-Contained) Attack

We outline (for brevity) here a self-contained attack against the Fuchsia/gVisor IPv4 ID generation algorithm. The full attack description and details can be found in App. D.

In this attack, we demonstrate that the underlying issue of hash collisions in H and, particularly, its small key space can be exploited independently, i.e., there is no need to rely on the PRNG attack or any other technique to obtain the source IPv4 address. We do not fully analyze the success/failure thresholds as in App. C for the straightforward attack (§ VI-A). Also, we do not attempt to reduce the dwell time to a value that allows large-scale deployment. Instead, we limit the discussion to showing that an attack is possible, and we report our initial measurements and provide a PoC which are not "production ready" – the expected failure rate is much higher than 10^{-6} and its dwell time is around one minute. Nevertheless, this shows that the problem is unrelated to the PRNG and requires addressing at the IPv4 ID level.

Our attack employs the loopback hash collision attack concept from Kol et al. [21]. The attack's goal is to find hash collisions of *loopback* IPv4 addresses, i.e., addresses in the range 127.0.0.0/8.

The attack uses a set R of several hundred public external IPv4 addresses on the attacker's machine, and a set I of several hundred loopback addresses. The attacker first finds a subset R' of external addresses which fall into unique buckets by removing colliding addresses from this list (using the technique in § VI-A).

Now, the attacker finds within R' a subset R'_I of remote addresses whose hashes are observed to collide with hashes of at least two loopback addresses in I. To do this, the attacker's script in the browser "sandwiches" a burst of connection attempts to the loopback addresses in I, between two bursts to R', and observing for which addresses in R' the IPv4 ID incremented by 5 (or more), since each loopback connection attempt increments the bucket counter by 2 (due to the TCP RST sent back). This is illustrated in Fig. 1.

For each address in R'_I, the attacker finds the colliding loopback addresses using a binary search, starting with I and halving it until only the colliding loopback addresses remain.

This provides the attacker with a list of loopback collisions. The attacker can extract the $hashIV$ from this list by enumerating over all 2^{32} key candidates and finding the best match, similar to the approach outlined in § VI-A. As a last step, using the found $hashIV$ and the external collisions in R, the attacker can enumerate over the 2^{32} possible IP_{SRC} values and find the one that yields the best match for the external collisions. This exposes the device's internal (private) IP address as well.

VII. Experiments and Results

As part of our research, we set up two PoC servers to demonstrate the attacks listed in this work. We allocated virtual machines in Microsoft Azure East US and Germany West Central regions. The virtual machine model used is Standard_E20as_v4 (20 vCPUs, 160 GiB memory). The camera-ready paper will include a link to the PoC source code.

We used the following Fuchsia devices to test the attacks:

- **Google Nest Hub Max** (smart home speaker+display), Amlogic T931 CPU (ARM 64bit architecture), running Fuchsia v11.20230306.135.
- **Google Pixelbook Go laptop**, Intel Core i7-8500Y CPU (x64 architecture), running Fuchsia v9.
- **Intel NUC mini-PC** model NUC8BEH, Intel Core i3-8109U CPU (x64 architecture) running Fuchsia v9.
- **Virtual device** (over QEMU for x64) running Fuchsia v9.

Table I Describes the networks used for the attacks. The table describes the network name, technology, IPv6 support, and whether the network modifies TCP source ports.

Table II describes the general results from our tests using the PoC servers. The table lists the attack, the target of the attack (kernel object), the number of bits in the target, the number of network packets involved in the attack, the dwell and compute times (average and maximum) in milliseconds, and the data exposed as a by-product of the attack, e.g., the device's internal (private) IP through the TCP attack over IPv4 and independent IPv4 ID attack.

In our experiments, we tracked 100% success with the PRNG seed extraction over IPv4 (51 experiments), the PRNG seed extraction over IPv6 (12 experiments), the independent $hashIV$ extraction (4 experiments) and the UDP port prediction (4 experiments). In the straightforward $hashIV$ experiment, we formally have one failure out of 12 experiments, but this case can be disregarded as discussed in § VIII-A. The difference in number of experiments is due to some experiments rely on IPv6, which is not available in all networks, and some experiments rely on prior seed extraction which necessitated running the PRNG seed extraction for them.

The $hashIV$ independent attack requires the browser to momentarily allocate a lot of memory to support a burst of TCP connections. Some devices cannot support this load. We experienced crashes on Google Nest and our virtual devices (running on QEMU).

We also measured PRNG invocations per day. On Google Nest we measured 14409/day and 29855/day averages over two time intervals, 10625/day on the Google Pixelbook and 13076/day on the Intel NUC device.

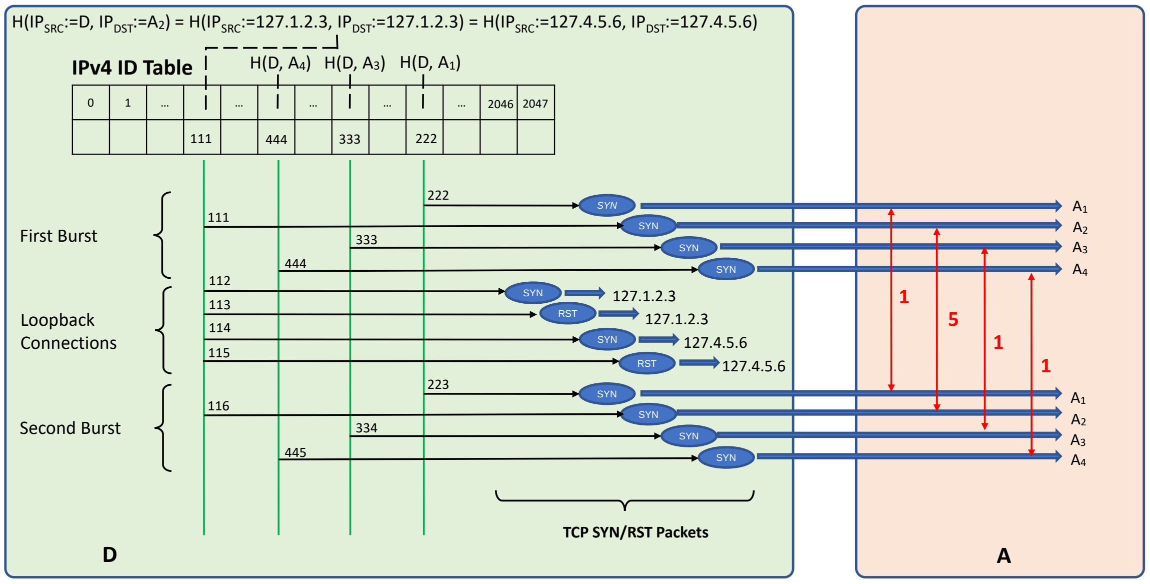

Figure 1. The first burst of SYN packets from source device D to attacker server A is sent to remote IP addresses $A_1, \ldots, A_4$ in the set R'. The attempted connection to A_2 causes cell $c = H(IP_{SRC} = D, IP_{DST} = A_2)$ of the IPv4 ID table to advance once. Next, packets are sent to all loopback addresses in I, including 127.1.2.3 and 127.4.5.6, whose hashes collide (at c). This collision causes the IPv4 ID table cell c to advance by four (two SYN packets and two RST packets). By monitoring the IP IDs of the second burst, the attacker observes that the IP ID of A_2 in the second burst has increased by five. Thus, the attacker determines that the IP address A_2 collides with two loopback addresses in I and needs to be included in R'_I.

Table I
NETWORKS

Network Name	Technology	IPv4/IPv6 Support	TCP Source Port
Bezeq	VDSL	Both	Override
Eduroam (HUJI)	Fiber (DWDM)	IPv4 Only	Intact
Triple C	VDSL	IPv4 Only	Override
Bezeq Fiber	Fiber	Both	Intact
Hot Cable	Cable	IPv4 Only	Override
Golan Telecom	Cellular	Both	Override
Partner	Cellular	IPv4 Only	Override

We took care of conducting a large portion of experiments on Chrome Incognito mode and noted no different behavior in comparison to regular browsing.

During the development of the PoC, Fuchsia was upgraded from v9 through to v12. We successfully tested our TCP-based PRNG seed extraction attack over IPv4 all four tested versions.

In all the above tests, the PRNG seed and the $hashIV$ hashing key were consistent per device across all tests for all devices (per single boot session) and differed between devices. A small portion of the PRNG seed and $hashIV$ extraction experiments was verified against the ground truth (via kernel logging of the actual values).

VIII. Discussion and Recommendations

A. Practical Considerations

While our PoC **compute time** was several seconds in the TCP and IPv4 ID experiments, we emphasize that the attack computation logic is embarrassingly parallel, and thus, by increasing the computing power by a certain factor, the attacker can cut down the compute time by the same factor.

We can also reduce the **memory consumption** by moving the Q, Q' and W' tables from the RAM to a fast storage (SSD), since these tables are only consulted a few times per device. Thus we can make do with only 32GiB RAM (table W) for IPv4 and negligible RAM for IPv6.

Predicted-but-unobserved IPv4 ID collisions – as explained in App. C, the upper bound on failure probability for the $hashIV$ straightforward attack relies on rejecting keys that produce unobserved collisions. Therefore, in this attack experiments, we also recorded whether the chosen key predicted unobserved collisions. In 11 out of 12 experiments, no collisions were predicted by the chosen key beyond the observed collisions. In one experiment over a cellular network, ten observed collisions matched, and five were predicted but unobserved. Packet loss over the cellular network can explain this high number. We cannot verify or refute this hypothesis

Table II
EXPERIMENT DESCRIPTIONS

Attack/ Vulnerability	Paper Section	Attack Object	Bits	Packets	Dwell Time [ms]	Compute Time [ms]	Additional Data Exposed
TCP fields (IPv4)	§ IV-B	PRNG Seed	31	4	7 (avg) 18 (max)	2937 (avg) 5776 (max)	Private IP address TCP connection counter
TCP fields (IPv6)	§ IV-C	PRNG Seed	31	2	2 (avg) 3 (max)	0.5 (avg) 1 (max)	TCP connection counter
UDP source port	§ V	Next source port	15.6	0	negligible	0.5 (avg) 1 (max)	
IPv4 ID (straightforward)	§ VI-A	$hashIV$	32	250	116 (avg) 170 (max)	5397 (avg) 5464 (max)	
IPv4 ID (independent)	§ VI-B	$hashIV$	32	Thousands	59109 (avg) 73075 (max)	20762 (avg) 20775 (max)	Private IP address

due to insufficient server-side logging. Thus, we regard this experiment as invalid yet warranting future inspection.

In theory, **organic noise** could interfere with our measurements. For the simplicity of the paper and the PoC, we assume that no additional TCP connection is established during our TCP measurements (4 packets/connections), and no additional PRNG invocations happen during the PRNG sampling (4 UDP port bindings). But since the former take up to 18ms and the latter takes a negligible amount of time (less than 1ms), this concern is not overly realistic. Indeed, our experiments, using Fuchsia devices in realistic scenarios all completed successfully. Moreover, it is trivial to adjust the tolerance of the TCP attack algorithms to cater for a few organic TCP connections in between our measurements. This can be done in step 4 of § IV-B by extending the equation $\widehat{sport} = 256 \cdot s_H + (s_L + 1)$ into $\widehat{sport} = 256 \cdot s_H + (s_L + 1 + \gamma)$ where γ is the number of organic TCP connections attempts between the 2nd and 3rd packets of the measurement. Now we can relax the assumption $\gamma = 0$ into $\gamma < G$ where G is a reasonable upper bound on the number of TCP connection attempts per e.g. 10ms, and enumerate over γ (G values).

In the IPv4 ID case, organic noise can indeed affect the measurement, but our key finding technique can withstand a reasonable amount of noise and indeed in our tests, we managed to find the correct key (up to the exception explained above which is probably not due to noise). Additionally, we can allow more noise as part of the technique (i.e. accept IP valued that differ by slightly more than 1 as a collision).

B. Additional Attacks

The IPv6 fragment ID generation algorithm is very similar to the IPv4 ID generation algorithm. As such, our straightforward attack against IPv4 ID can be adapted, under certain conditions, to IPv6. However, it should be noted that the IPv6 fragment ID is only generated when fragmentation is needed, and this condition is not trivial to trigger from Javascript code on the browser (because WebRTC packets are short). On the other hand, it is trivial to trigger outside the browser, e.g., using long ICMPv6 Echo requests remotely. We have not experimented with IPv6 ID.

In the paper we focused on device tracking, but the vulnerabilities we uncovered can be exploited for **additional attacks and use cases** (which we do not demonstrate):

- **TCP initial sequence number prediction** is a special case of a TCP sequence number prediction attack [46], which can be used to counterfeit TCP packets.
- In general, **predictable UDP source ports** can be exploited to mount an effective DNS cache poisoning attack [47]. To be vulnerable to our attack, the DNS software has to use the underlying operating system to select UDP source ports (this is **not** the case with Fuchsia's stub DNS resolver; however, there may be other gVisor use cases in which this applies).
- **TCP source port predictability** can expedite a TCP blind reset attack [48].
- Our attacks on TCP/IPv4 and the IPv4 ID **disclose the device's internal IP address**, even if the device is behind a NAT. This violates PCI DSS 4.0 Requirement 1.4.5 [49].
- Our attack on the IPv4 ID generation algorithm can be used to find IP addresses that collide (served from the same bucket) in the IPv4 ID table, which can be used to **predict IPv4 ID values** across these addresses. This in turn facilitates DNS cache poisoning attacks [50], traffic analysis attacks [51], [52], [53] and TCP hijacking attacks [54].
- In general, when a PRNG is used to generate sensitive data, having a **predictable PRNG** may result in security vulnerabilities.

For example, we describe herein a TCP hijacking attack against an HTTP proxy server running on top of a gVisor network stack. A remote attacker (as a client) first forces the HTTP proxy to connect to the attacker IP addresses (by sending it HTTP requests to be forwarded to these addresses), obtaining the PRNG seed and also measuring the RTT between the attacker client and the server. Then the attacker client forces the proxy to load a page from `www.example.com`, while DDoSing the genuine `www.example.com` server. The attacker can predict the exact time the proxy sends the TCP SYN packet to `www.example.com` (using the RTT measurement, and assuming 1ms granularity). Thus the attacker can predict the TCP source port and ISN (the latter – up to $\approx 16,000$ candidates). With this, the attacker can spoof the TCP+ACK response from `www.example.com`, "establish" the TCP circuit and poison the proxy's web cache. A similar scenario can be described with the attacker triggering the proxy from a malicious Javascript running inside a browser. Testing these scenarios, however, was out of scope for this research project.

C. Analysis of the Vulnerabilities: Individually and in Concert

The weakness of Jenkins' One-at-a-Time Hash function alone leads to predictable TCP ISN values because, as we show in the TCP attacks, J_{ISN} can be extracted (together with J_{TS}) without any assumptions on the PRNG or any other components of the system. With J_{ISN}, it is possible to predict TCP ISN (up to the clock value) if the 4-tuple values are known. We recommend using cryptographically strong hash functions for generating protocol header fields, with sufficiently large hashing keys. For example, OpenBSD uses SHA2 for TCP ISN and TCP timestamp generation.

The Go PRNG advancement algorithm weakness alone suffices for the UDP port prediction attack, as this attack does not assume anything about the PRNG seeding. According to its documentation, the Go `rand` package "should not be used for security-sensitive work" [55]. We recommend using a cryptographically strong PRNG for populating network protocol header fields. Seeding the PRNG should be done with sufficient entropy. For example, Linux uses ChaCha20 for its kernel PRNG, indicating that this approach is sound with regards to performance.

The Go PRNG small effective seed space size alone suffices for the UDP seed extraction attack since it does not make any assumption on the PRNG advancement mechanism. This is CPU and RAM intensive, and as such may not be economic in practice.

The use of a global counter as part of the TCP source port generation contributes to our TCP attacks. We recommend completely randomizing the source port as prescribed in "Algorithm 1/2" [40]. This is the approach taken by both FreeBSD and OpenBSD, thus it may be considered a reasonable practice.

The combination of IPv4/IPv6 ID small hashing key space together with the hash table size and deterministic update scheme suffices to extract the key regardless of the hash function, as demonstrated in § VI-B. This attack does not rely on obtaining the internal (private) IP address. However, it does require excessive dwell time and as such does not fit many device tracking scenarios. In App. E, we outline an extended attack that relies solely on collisions, without any assumptions on the hash function algorithm and key size. To thwart attacks based on hash collisions, we recommend generating completely random values for IPv4 ID and IPv6 ID. This may yield a higher rate of ID collisions, especially for IPv4, but since this approach has already been employed by XNU (iOS, macOS, etc.) and NetBSD for IPv4 ID, and by Linux and NetBSD for IPv6 ID, for quite a while, it may be considered a reasonable practice.

As can be seen above, the **root causes** of the vulnerabilities we found are weaknesses in the cryptographic algorithms used in Fuchsia/gVisor, as well as fundamental security issues that stem from hash collisions (in the Fuchsia NetStack3 TCP source port algorithm – App. A, and in the gVisor IP ID generation algorithm – App. E). The latter may indicate that the construction of an array of counters indexed by a hash function is inherently insecure when the input to the hash function can be attacker-controlled. Replacing these constructs is not straight-forward, since other approaches entail a different balance between security and functionality.

Each of the above five vulnerabilities stems from an individual, independent root cause. When considered separately, each attack may appear to be minor or less feasible. However, in a **holistic analysis**, when all the attacks are considered together, we show the interplay of the vulnerabilities yields a powerful PRNG seed extraction and IP ID hash key extraction attacks, which are very feasible and economic.

D. Partial Mitigation

Extending the PRNG seed to 64 bits[6] **(or more)** has a limited effect on the attacks. The last step of the TCP/IPv4 attack (§ IV-B step 5) requires a table W which is generated by enumerating over the seed space. Thus step 5 becomes impractical and the seed cannot be extracted. However, steps 1-4 are not affected, and as such, the attacker can still calculate J_{TS} and J_{ISN} which can facilitate TCP hijacking attacks. Moreover, in the TCP/IPv6 attack (§ IV-C), while calculating the table W' is again impractical and thus the seed cannot be obtained, it is still possible to extract S_{TS} which is a 32 bit secret independent of the network and local IP address, and as such can be used as a device ID (and similarly, S_{ISN} and S_{PS}). Finally, the UDP source port prediction technique (§ V) is oblivious to the PRNG seed size, and the attacks against IP ID (§ VI) are unrelated to the PRNG and are thus unaffected by the choice of seed size. The only attack completely thwarted by this is the seed extraction via UDP source ports (§ IV-D) which is already compute-heavy to begin with.

Switching the Go PRNG to a stronger one has a minor effect on the overall security of Fuchsia/gVisor, since the only attack that exploits the PRNG logic is the UDP source port prediction (§ V).

Even if we additionally employ **cryptographically strong hash functions**, some attacks will still be applicable: the IPv6 extraction of S_{TS} (and S_{ISN}, S_{PS}) is still feasible (with a 32-bit real-time enumeration) as well as the IP ID attack we describe in App. E.

IX. Conclusion

In this paper, we holistically reviewed the security of algorithms that populate various protocol header fields of Google Fuchsia's gVisor-based network stack. Fuchsia is considered to

[6]The original PRNG seeding code takes a 64-bit seed and "truncates" it to a 31-bit effective seed. It is trivial to simply omit the truncation step.

be the future operating system of smartphones, tablets, and IoT devices, and it is already deployed to millions of shipped Nest Hub and Nest Hub Max smart display+speaker devices. The gVisor kernel is used in several key Google Cloud offerings. Therefore, this research has a clear impact at present and when looking forward.

Our findings span multiple protocol headers (TCP Initial Sequence Number, TCP timestamp, TCP source port, UDP source port, and IPv4/IPv6 ID), demonstrating that the gVisor stack suffers from numerous security issues. We identify the root causes as the use of a weak PRNG (Go's built-in PRNG) which has a too-small seed space and an insecure advancement mechanism, the use of the weak Jenkins "one-at-a-time" hash function, and the use of a hash-based IP ID generation mechanism that is prone to easy hash collision detection, and makes use of a too-small hashing key.

When combined, these vulnerabilities allow an attacker to extract the PRNG seed and the IP ID hashing key, which can facilitate various attacks. We describe multiple device tracking techniques based on our findings. We demonstrate our tracking attacks with two PoC servers in two continents. We conduct experiments with these PoC servers in real-life settings, from multiple Fuchsia devices, over multiple networks (including IPv4 vs. IPv6), and browsing modes (regular/private). We report an excellent success rate and dwell time, which proves that our device-tracking attacks are practical and scalable.

Finally, we provide recommendations on how to generate the affected protocol header fields securely. We disclosed our report and our recommendations to the Fuchsia and gVisor security teams.

This research demonstrates the importance of holistic security analysis of network stacks, especially with respect to new or under-scrutinized kernels. Many works only look at a single protocol header field (or algorithm), and as such may miss the interplay between "minor" vulnerabilities that may yield much more powerful attacks. However, when the network stack is reviewed as a whole, such attacks can be revealed.

X. Vendor Status

We reported our findings to Google on October 4th, 2023. Google assigned CVE-2024-10026, CVE-2024-10603 and CVE-2024-10604 to track these vulnerabilities. Per our recommendations, Google released the following patches for gVisor and Fuchsia:

- gVisor commit 83f7508 – Replace the insecure Go default PRNG with a reference to Go's `crypt/rand` secure PRNG for the network protocol fields, CVE-2024-10026.
- gVisor commits f956b5a, e54bfde – Switch the hash functions used in TCP ISN and TCP TS generation to (truncated) SHA2/256 with 16 bytes hashing key, CVE-2024-10026.
- gVisor commit cbdb2c6 – Randomize the TCP source port per RFC 6056 Alg. 1 (using a secure PRNG), CVE-2024-10603.
- gVisor commits 6262b00, 505e1fd, 5d2bf25 – Randomize the IPv4 and IPv6 ID (using a secure PRNG), CVE-2024-10603.
- Fuchsia commits ac2fca3, c436a1d – Port the above gVisor patches to the Fuchsia code branches.
- Fuchsia commits a3c17a4, 40e7fbc – Randomize the NetStack3 TCP and UDP source port per RFC 6056 Alg. 1 (a remediation against the attack in App. A), CVE-2024-10604.

The gVisor patches are included in gVisor build release-20231204.0 and in Fuchsia version 16 (released February 14th ,2024). The Fuchsia-specific patch (NetStack3's TCP and UDP source port algorithms) is included in Fuchsia version 20 (released June 4th, 2024).

Acknowledgement

We deeply thank Benny Pinkas for his review of the research in its early stages and for his feedback and review of multiple drafts of this paper. We are grateful to the anonymous reviewers of NDSS 2025 for their valuable feedback and suggestions. We thank the anonymous NDSS 2025 shepherd for shepherding our paper.

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 1071/23).

References

[1] T. F. Team, "Learn about Fuchsia," https://fuchsia.dev/fuchsia-src/get-started/learn-fuchsia, 2022.
[2] The gVisor Team, "gVisor – The Container Security Platform," https://gvisor.dev/, 2023.
[3] E. Gonzalez, "Google's mysterious OS: Fuchsia," http://androiddeveloper.galileo.edu/2019/09/16/googles-mysterious-os-fuchsia/, Sep. 2019.
[4] R. Amadeo, "Google's "Fuchsia" smartphone OS dumps Linux, has a wild new UI," https://arstechnica.com/gadgets/2017/05/googles-fuchsia-smartphone-os-dumps-linux-has-a-wild-new-ui/, May 2017.
[5] L. Booker, "Google Is Working On 'Fuchsia', Its Own, Non-Linux Based Operating System For 'Modern' Devices," https://gizmodo.com.au/2016/08/google-is-working-on-fuchsia-its-own-non-linux-based-operating-system-for-modern-devices/, Aug. 2016.
[6] A. Sinicki, "What is Google Fuchsia? Is this the new Android?" https://www.androidauthority.com/what-is-google-fuchsia-os-847784/, Jun. 2019.
[7] J. Soni, "Is Samsung looking to replace Android with Fuchsia OS?" https://www.techradar.com/news/is-samsung-looking-to-replace-android-with-fuchsia-os, Dec. 2021.
[8] Indo-Asian News Service, "Google experimenting Fuchsia OS with Honor smartphone: Report," https://newsd.in/google-experimenting-fuchsia-os-with-honor-smartphone-report/, Aug. 2018.
[9] M. Rahman, "Google is bringing Fuchsia OS to Android devices, but not in the way you'd think," https://www.androidauthority.com/microfuchsia-on-android-3457788/, 2024.
[10] A. Li, "Fuchsia starts rolling out to 2nd-gen Nest Hub," https://9to5google.com/2023/05/23/nest-hub-2nd-gen-fuchsia/, May 2023.
[11] ——, "Google completes Fuchsia rollout for Nest Hub Max," https://9to5google.com/2022/08/24/nest-hub-max-fuchsia-rollout/, Aug. 2022.
[12] Strategy Analytics, "Another Record Quarter for Smart Speakers in 3Q21, Though Supply Chain Woes are on the Horizon," https://www.businesswire.com/news/home/20211220005569/en/Strategy-Analytics-Another-Record-Quarter-for-Smart-Speakers-in-3Q21-Though-Supply-Chain-Woes-are-on-the-Horizon, Dec. 2021.
[13] T. gVisor Team, "gvisor," https://github.com/google/gvisor, 2021.
[14] Wikipedia Authors, "gvisor," https://en.wikipedia.org/wiki/GVisor, 2019.

[15] J. Wilander, "Full Third-Party Cookie Blocking and More," https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/, Mar. 2020.

[16] M. Mihajlija, "Prepare for phasing out third-party cookies," https://developer.chrome.com/en/docs/privacy-sandbox/third-party-cookie-phase-out/, 2023.

[17] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros, "Web Tracking: Mechanisms, Implications, and Defenses," *CoRR*, vol. abs/1507.07872, 2015. [Online]. Available: http://arxiv.org/abs/1507.07872

[18] A. Klein and B. Pinkas, "From IP ID to Device ID and KASLR Bypass," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1063–1080.

[19] A. Klein, "Subverting stateful firewall with protocol states," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, 27 February - 3 March 2022*. The Internet Society, Feb. 2022.

[20] ——, "Cross Layer Attacks and How to Use Them (for DNS Cache Poisoning, Device Tracking and More)," in *2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 927–944.

[21] M. Kol, A. Klein, and Y. Gilad, "Device tracking via Linux's new TCP source port selection algorithm," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6167–6183.

[22] Y. Li, Y. Chen, S. Ji, X. Zhang, G. Yan, A. X. Liu, C. Wu, Z. Pan, and P. Lin, "G-fuzz: A directed fuzzing framework for gvisor," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2023.

[23] D. Katz, "Attacking Go's Lagged Fibonacci Generator," https://www.leviathansecurity.com/media/attacking-gos-lagged-fibonacci-generator, Nov. 2022.

[24] J. Berger, A. Klein, and B. Pinkas, "Flaw Label: Exploiting IPv6 Flow Label," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1344–1361.

[25] Forescout Research Labs, "Number:Jack – Weak ISN Generation in Embedded TCP/IP Stacks," https://www.forescout.com/resources/numberjack-weak-isn-generation-in-embedded-tcpip-stacks/, Feb. 2021.

[26] D. dos Santos, S. Dashevskyi, A. Amri, J. Wetzels, A. Karas, S. Menashe, and D. Vozniuk, "Infra:Halt – Jointly discovering and mitigating large-scale OT vulnerabilities," https://www.forescout.com/resources/infrahalt-discovering-mitigating-large-scale-ot-vulnerabilities/, Aug. 2021.

[27] D. dos Santos, S. Dashevskyi, J. Wetzels, and A. Amri, "AMNESIA:33 – How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices," https://www.forescout.com/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/, Dec. 2020.

[28] B. Seri, G. Vishnepolsky, and D. Zusman, "Urgent/11 – Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS," , 2019.

[29] M. Kol, A. Schon, S. Oberman, A. Dotan, A. Zagrebin, and Y. Shapiro, "Ripple20 – 19 Zero-Day Vulnerabilities Amplified by the Supply Chain," https://www.jsof-tech.com/disclosures/ripple20/, 2020.

[30] D. Roesler, "Demo," https://diafygi.github.io/webrtc-ips/, 2015.

[31] Q. Wang, "Issue 878465: mDNS service for IP handling in WebRTC," https://bugs.chromium.org/p/chromium/issues/detail?id=878465#c22, Oct. 2020.

[32] A. Klein and B. Pinkas, "DNS Cache-Based User Tracking," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, Feb. 2019.

[33] M. Schwarz, F. Lackner, and D. Gruss, "Javascript template attacks: Automatically inferring host information for targeted exploits," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[34] H. Venugopalan, K. Goswami, Z. A. Din, J. Lowe-Power, S. T. King, and Z. Shafiq, "Centauri: Practical rowhammer fingerprinting," 2023.

[35] M. M. Ali, B. Chitale, M. Ghasemisharif, C. Kanich, N. Nikiforakis, and J. Polakis, "Navigating murky waters: Automated browser feature testing for uncovering tracking vectors," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[36] G. Patat, M. Sabt, and P. Fouque, "Your DRM can watch you too: Exploring the privacy implications of browsers (mis)implementations of widevine EME," *Proc. Priv. Enhancing Technol.*, vol. 2023, no. 4, pp. 306–321, 2023.

[37] B. Jenkins, "A Hash Function for Hash Table Lookup," http://www.burtleburtle.net/bob/hash/doobs.html, Nov. 2013.

[38] H. Aptroot, "Simplify the inverse of z = x ^ (x << y) function," https://stackoverflow.com/questions/31521910/simplify-the-inverse-of-z-x-x-y-function, Jul. 2015.

[39] W. Schultz, "Cracking Go's PRNG For Fun and (Virtual) Profit," https://will62794.github.io/security/hacking/2017/06/30/cracking-golang-prng.html, Jun. 2017.

[40] M. V. Larsen and F. Gont, "Recommendations for Transport-Protocol Port Randomization," RFC 6056, 2011.

[41] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630.

[42] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.

[43] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Boston: Addison-Wesley, 1997.

[44] devteam_...@netop.com, "Issue 825576: RTCPeerConnection objects are released too slowly and reallocating causes exception: Cannot create so many PeerConnections," Mar. 2018. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=825576

[45] B. Jenkins, "lookup3.c," http://burtleburtle.net/bob/c/lookup3.c, May 2006.

[46] S. M. Bellovin, "Security problems in the tcp/ip protocol suite," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 2, p. 32–48, Apr. 1989. [Online]. Available: https://doi.org/10.1145/378444.378449

[47] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "Dns cache poisoning attack reloaded: Revolutions with side channels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1337–1350.

[48] M. Luckie, R. Beverly, T. Wu, M. Allman, and K. Claffy, "Resilience of Deployed TCP to Blind Attacks," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–26.

[49] PCI Security Standards Council, "Payment Card Industry Data Security Standard Requirements and Testing Procedures Version 4.0," https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0.pdf, Mar. 2022.

[50] A. Herzberg and H. Shulman, "Fragmentation Considered Poisonous," *CoRR*, vol. abs/1205.4011, 2012. [Online]. Available: http://arxiv.org/abs/1205.4011

[51] X. Zhang, J. Knockel, and J. R. Crandall, "ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path." in *INFOCOM*. Los Alamitos, CA, USA: IEEE, 2018, pp. 2069–2077.

[52] W. Chen, Y. Huang, B. F. Ribeiro, K. Suh, H. Zhang, E. de Souza e Silva, J. Kurose, and D. Towsley, "Exploiting the ipid field to infer network path and end-system characteristics," in *Passive and Active Network Measurement*, C. Dovrolis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 108–120.

[53] G. Alexander, A. M. Espinoza, and J. R. Crandall, "Detecting TCP/IP Connections via IPID Hash Collisions," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, pp. 311 – 328, Oct. 2019.

[54] X. Feng, C. Fu, Q. Li, K. Sun, and K. Xu, "Off-path tcp exploits of the mixed ipid assignment," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1323–1335.

[55] The Go Project Team, "rand package," https://pkg.go.dev/math/rand, 2014.

[56] T. F. Team, "port_alloc.rs," https://cs.opensource.google/fuchsia/fuchsia/+/main:src/connectivity/network/netstack3/core/src/algorithm/port_alloc.rs, 2023.

[57] B. O'Neill, "The classical occupancy distribution: Computation and approximation," *The American Statistician*, vol. 75, no. 4, pp. 364–375, 2021.

[58] T. Rigoudy and M. West, "Private network access," https://wicg.github.io/private-network-access/, Jun. 2021.

Appendix A
Device Tracking Based on the TCP and UDP Port Assignment in Fuchsia's NetStack3

Fuchsia's source code tree contains an alternative dedicated network stack called "NetStack3", written entirely in Rust. It is not used by default – special compilation options and procedures must be applied for the Fuchsia operating system to use this alternative implementation. Nevertheless, we analyzed NetStack3 as well and found its TCP and UDP source port assignment algorithm vulnerable. Note that this analysis is based on source code review; we did not investigate a live system or set up a PoC to test a live system.

In essence, NetStack3 implements algorithm 4 of RFC 6056 [40] ("DHPS" in Kol et al.'s terminology [21]) as-is, with table size $T = 20$, and using a truncated HMAC-SHA256 hashing function [56]. As such, it is vulnerable to the attack approach of [21]. Since the necessary condition $T - (1 + \frac{1}{2}\ln T)\sqrt{T} - \frac{1}{4}\ln T \geq 2\ln N - \ln 2$ [21, Appendix A.4] does not hold for the desired tracked population size ($N = 1,000,000$), the exact attack techniques described in [21] cannot be applied as they are. However, it is easy to see that they can be adapted for the case $T = 20$. For the second phase of the attack, the attacker needs to iterate over L loopback addresses, where $L \geq T$ will be calculated per the population size. Suppose $t \leq T$ buckets are covered for a device D. The probability of a random device D' to yield the same signature is (by the same arguments of [21]) $\frac{T}{T}\frac{T-1}{T}\cdots\frac{T-(t-1)}{T}\frac{1}{T^{L-t}} = \frac{T(T-1)\cdots(T-(t-1))}{T^L}$. It is easy to see that the worst case (i.e. highest probability for signature collision) is when t is maximal, i.e. $t = T$. In the worst case, the probability is $\frac{T!}{T^L}$. To obtain an average of up to one signature collision, we require, for N devices, that $\frac{T!}{T^L} \leq \frac{1}{\binom{N}{2}}$. This finally yields $L \geq \log_T \binom{N}{2} T!$. For $N = 1,000,000$ we thus have $L \geq 23.12$. Therefore, $L = 24$ is the number of loopback addresses needed in the second phase, much smaller than the number needed in the original attack against Linux. Hence, the attack against Fuchsia's NetStack3 may be much faster than the Linux attack in [21].

Appendix B
Fast f Calculation

For the rainbow table time-memory trade-off of inverting a function f, it is necessary to calculate f quickly and efficiently. In our case, $f(s, r)$ is a hash g of the sequence of $n = 4$ consecutive UDP source ports generated from the PRNG seeded with s after r PRNG steps. Naïvely seeding the PRNG and rolling it forward r steps is problematic on two accounts: the seeding process is very expensive in terms of CPU operations, and rolling the PRNG forward r steps is $O(K)$ in CPU operations, while ideally, f calculation would be in $O(1)$.

We now present a technique for a fast calculation of f. For this, we employ the concept of time-memory trade-off twice. We reduce the seeding procedure computation time by building, offline, a memory map from all 2^{31} seeds to their initial internal PRNG state ($607 \cdot 8$ bytes). We reduce the computation time of rolling the PRNG multiple steps by building, offline, a map from all K PRNG offsets to the representations of the next $n = 4$ raw PRNG outputs as linear combinations of the initial words in the initial PRNG state (each cell in the map is thus $607 \cdot n$ 64-bit integers). In total, we need $(607 \cdot 8)(2^{31} + K \cdot n)$=9.7TiB of RAM.

The calculation of $f(s, r)$ then becomes a table lookup for the initial state after seeding with s, a table lookup for the suitable linear combinations after r steps, and calculating n linear combinations (607 additions and multiplications each – a total of $607 \cdot n$ basic operations). Thus the leading term in the CPU time is $607 \cdot n$ (technically n is $O(logK)$ but this can be practically ignored).

Appendix C
Calculation of Failure Probability for $hashIV$ Extraction

For § VI-A, we define the key extraction algorithm as "finding the key which has the most matches to the pairs found, and no collisions that were not found.". A failure is a situation wherein one of the "false" $2^{32} - 1$ keys obtains as many matches or more than the correct key.

We define:

- L – the number of IP addresses on the attacker's machine.
- *the number of independent collisions (in a collision set)* is the size of a minimal subset of collisions from which the entire set of collisions can be inferred. For example, for a collision set (pairs of colliding addresses) $\{(a, b), (b, c), (c, a)\}$ of size 3, one minimal collision set (not the only one) is $\{(a, b), (b, c)\}$ because (c, a) can be inferred from (a, b) and (b, c). Therefore, the number of independent collisions for the above set is 2.
- $P_T(m)$ – the probability of the correct key to generate exactly m independent collisions among the L IP addresses. Assuming all collisions are detected, this will yield m pair matches on the observed data.
- $P_F(f)$ – the probability of exactly f false collisions, i.e. collisions that are generated "randomly", by having the bucket counters randomly match (probability 2^{-16}). For simplicity, we allow a true collision also to be a false collision (with probability 2^{-16}). In reality, this is considered only as a true collision (i.e., the real total collision count is not necessarily $m + f$, but rather $\leq m+f$), thus decreasing the failure rate. In other words, this simplification increases the failure rate, so the failure rate we calculate here is an upper bound. However, it should, in reality, be quite tight.
- $Q_F(m, f)$ – the probability of *any* false key (one of the $2^{32} - 1$) to match at least m collisions (out of the m true collisions and the f false collisions).

In general, we have:

$$P_{FAILURE} = \sum_{f=0}^{L-1} P_F(f) \sum_{m=0}^{L-1-f} P_T(m) Q_F(m, f)$$

For P_T, we need to calculate the probability of L addresses to have precisely m independent collisions. This is equivalent to the occupancy problem (e.g., [57]). The occupancy problem asks for the distribution of occupied (non-empty) buckets when throwing $\tilde{n}$ balls into $\tilde{m}$ bins. The answer is the classical occupancy distribution (from [57]):

$$Occ(\tilde{k}|\tilde{n},\tilde{m}) = \frac{1}{\tilde{m}^{\tilde{n}}}\binom{\tilde{m}}{\tilde{k}}\sum_{i=0}^{\tilde{k}}\binom{\tilde{k}}{i}(-1)^i(\tilde{k}-i)^{\tilde{n}}$$

In our case, we ask for the probability of having $\tilde{k} = L - m$ occupied bins when throwing $\tilde{n} = L$ balls to $\tilde{m} = 2048$ buckets, therefore:

$$P_T(m) = Occ(L-m|L, 2048)$$

This is strictly defined for $\max(0, L-2048) \leq m \leq L-1$, but we can extend it to 0 elsewhere.

For $P_F(f)$, the number of false collisions, we similarly use

$$P_F(f) = Occ(L-f|L, 2^{16})$$

Suppose we have $L-m$ occupied (non-empty) buckets B_i, and we have n_i IP addresses in B_i, so that $\sum_i n_i = L$. For a random hashing key to generate hash values that exactly matches this structure, we look at the first IP address in each bucket. When building the buckets for the random key, we start with all empty buckets. The first IP address can pick any bucket; thus, the probability to succeed is $\frac{2048}{2048}$. The second IP address (which is first in its bucket) has probability $\frac{2048-1}{2048}$ since it must not hit the first bucket, and so forth. So the success probability of all first IP addresses in their buckets is $\prod_{i=0}^{L-m-1}(1-\frac{i}{2048})$. The remaining IP addresses must go each to its bucket, so each one has probability $\frac{1}{2048}$ to hit the correct bucket. Therefore:

$$P^L_{RANDOM}(m) = \frac{\prod_{i=0}^{L-m-1}(1-\frac{i}{2048})}{2048^m}$$

Interestingly, this probability does not depend on the "structure" ($\{n_i\}$) of the fingerprint, which means that $P^L_{RANDOM}(m)$ is indeed well defined.

Given m true collisions and f false collisions, the total number of collisions is $m+f$. The probability of an arbitrary key to match less than m collisions is $1 - \sum_{i\geq m}\binom{m+f}{i}P^L_{RANDOM}(i)$.

Therefore, the probability of all $2^{32}-1$ false keys to all match less than m pairs is

$$\Big(1-\sum_{i=m}^{m+f}\binom{m+f}{i}P^L_{RANDOM}(i)\Big)^{2^{32}-1}$$

Finally, $Q(m,f)$ is the probability of the complementary event, the probability of at least one false key to match at least m collisions:

$$Q_F(m,f) = 1-\Big(1-\sum_{i=m}^{m+f}\binom{m+f}{i}P^L_{RANDOM}(i)\Big)^{2^{32}-1}$$

In Fig. 2 we plot $P_{FAILURE}$ for $200 \leq L \leq 330$ and observe that $P_{FAILURE}$ drops below 10^{-6} at $L = 250$ which guarantees that with $L = 250$ IP addresses there is less than a one in a million chance for a failure in obtaining the correct $hashIV$ for a device.

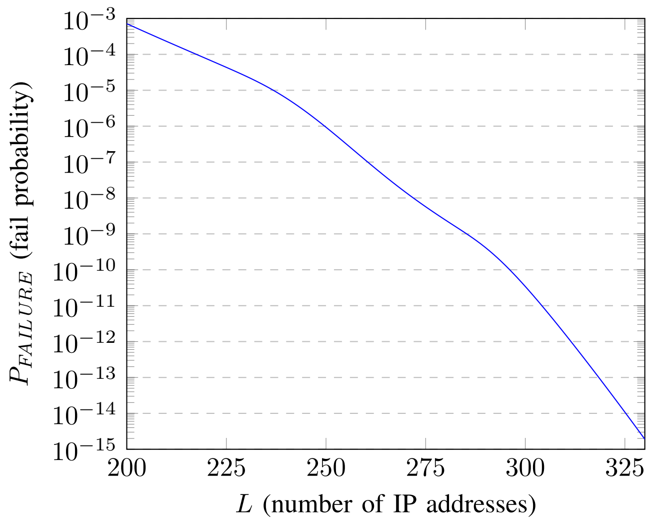

Figure 2. Failure probability per L

Appendix D
An Independent Attack against IPv4 ID

In this attack, we demonstrate that the underlying issue of hash collisions in H and, particularly, its small key space can be exploited independently, i.e., there is no need to rely on the PRNG attack or any other technique to obtain the source IP address.

Our attack employs the loopback hash collision attack concept from Kol et al. [21]. We look at hash collisions of *loopback* IPv4 addresses, i.e., addresses in the range 127.0.0.0/8. Interestingly, in Fuchsia/gVisor, when TCP/UDP traffic is destined to an address of the form 127.x.y.z, the source address is also set to 127.x.y.z (in contrast, in Linux, the source address is set to 127.0.0.1). Thus, when sending a TCP SYN packet to a closed TCP port on 127.x.y.z, the source IP is set to 127.x.y.z, and the kernel responds with RST from 127.x.y.z to 127.x.y.z. Therefore, the SYN and the RST consume their IP IDs from the same bucket. Armed with this knowledge, we show how an attacker can extract $hashIV$. The attack is facilitated via a list I of randomly chosen loopback addresses used by the Javascript code, as explained below. Later we will see that the optimal $|I|$ is slightly below 500.

The method of finding a list of loopback collisions is as follows. The attacker uses a set R (in our case $|R| = 250$)[7] of public external IPv4 addresses on the attacker's machine. The attacker first finds a subset of external addresses IP_r which fall into unique buckets (i.e., have unique $H(IP_{SRC}, IP_r, transport, hashIV)$ values where $IP_r \in R$ and $transport = 6$ is the transport protocol number for TCP) by removing colliding addresses from this list. This is done by sending $|R| = 250$ TCP SYN packets to the attacker's remote addresses using webRTC TURN and filtering out those that collide (i.e., addresses whose packets' IP IDs

[7]We chose $|R| = 250$ to reuse the 250 IPv4 addresses allocated for the basic attack in § VI-A. In practice, the attacker can use more IPv4 addresses.

are one value apart). Denote the resulting subset of external IP addresses whose hashes are unique by R'. The distribution of $|R'|$ is $Prob(|R'| = r) = occ(r, |R|, 2048)$ where occ is the occupation distribution [57].

Now, the attacker finds within R' a subset of remote addresses whose hashes are observed to collide with hashes of at least two loopback addresses in I, i.e., remote addresses whose hashes collide with loopback address hash collisions in I. In expectancy, there are $|R'|/2048 \cdot \binom{|I|}{2}/2048$ such pairs of addresses in I. The attacker can observe such collisions using the following method. First, the attacker's Javascript instructs the browser to attempt a TCP connection (send a TCP SYN packet), e.g., using WebRTC TURN, with all addresses in R. Next, the attacker does the same with the loopback list I (as reported in [21], the Private Network Access [58] restriction does not apply to WebRTC; hence, this traffic is allowed by the browser). Observe that loopback address hash collisions mean that respective buckets have been incremented by at least five. This stems from the two RST packets sent by the kernel over the loopback interface, in addition to the three other sent SYN packets – one to the remote address and two to loopback ones. We denote these remote addresses with IP IDs advanced by five as R'_I. Fig. 1 illustrates this process.

With R'_I, the attacker starts an "extended" binary search for loopback hash collisions in the space of loopback addresses I. The search narrows down the candidate list by recursively repeating the following. Let $I' \subseteq I$. The search starts with $I' = I$. In each search step, The attacker's Javascript attempts to connect (TCP) to every address in I' and then to every address in R'_I. The attacker then counts the total number of loopback addresses that collide with R'_I: $c_{I'} = \sum_{R'_I} \frac{\Delta IPID-1}{2}$. If $c_{I'} = 0$, the search is discontinued for the I' considered. Otherwise, if $|I'| = 1$, the search algorithm reports $IP \in I'$ and its corresponding external IP address (whose IP ID matches the loopback address found) as part of a hash collision and concludes the search for the branch considered. For any other result, the attacker splits I' into two halves, and the search continues recursively on each half.

This provides the attacker with a list of loopback collisions. The attacker can extract the $hashIV$ from this list by enumerating over all 2^{32} key candidates and finding the best match, similar to the approach outlined in § VI-A. For this technique to work in practice (as explained above, we do not provide failure thresholds in this case, so we use the expected number to estimate the practicality), we need the (expected) number of observable collisions in I, $|R'|/2048 \cdot \binom{|I|}{2}/2048$, to "overcome" the $2^{32} - 1$ false positive keys. Since each pair contributes 11 bits of information, we need $|R'|/2048 \cdot \binom{|I|}{2}/2048 > 3$. Consider that $|R'|$ is slightly smaller than $|R|$, we can approximate this as $|R|/2048 \cdot \binom{|I|}{2}/2048 \geq 4$, which (for $|R| = 250$) yields $|I| \geq 367$. In our experiment, we used $|I| = 490$, which yields 7.14 observable loopback address collisions, in expectation. $|I| = 490$ was chosen to maximize the number of loopback collisions while keeping a safe distance from the maximum concurrent WebRTC connections in Chrome (500). It should be noted that using loopback IP addresses overcomes the problem of not observing the device's source (private) IP address.

As a last step, using the found $hashIV$ and the external collisions in R, the attacker can enumerate over the 2^{32} possible IP_{SRC} values and find the one that yields the best match for the external collisions. This exposes the device's internal (private) IP address as well.

Appendix E
H-Agnostic Independent Attack against IPv4 ID

We now outline an an extension of the attack in App. D, which is H-agnostic, i.e. it assumes nothing on H. This demonstrates that the gVisor IPv4 ID generation algorithm is inherently vulnerable to device tracking, regardless of the choice of hash function H.

Recall that during the attack in App. D, the attacker obtains colliding loopback addresses. For $|R| = 250$, the attacker obtained 7.14 pairs. The choice of $|R| = 250$ was due to the availability of IPv4 addresses for our PoC, but in reality we can choose a number close to 500, e.g. $|R| = 490$, which yields $|R'| = 373$, in expectancy. Substituting this in the calculation for expected number of loopback collision pairs (assuming $|I| = 490$), we get $N = 10.7$ loopback collisions.

Note that when the device connects from different networks, different sets of pairs will be calculated – this is due to differing IP_{SRC} which induce different "windows" into the colliding set. So the set of pairs cannot be used as a device ID as-is. However, at the same time, the set of colliding pairs is always a subset of *all* colliding pairs in I. In expectancy, there are $\binom{|I|}{2}/2048 = 58.5$ such pairs. Therefore, in expectancy, there are $N^2/58.5 \approx 2$ pairs in the intersection of two device measurements (each one with say N loopback collision pairs), whereas the probability of measurements from two different devices to have two or more pairs in their intersection is approximately:

$$1-(1-N/\tbinom{2048}{2})^N - N \cdot N/\tbinom{2048}{2}(1-N/\tbinom{2048}{2})^{N-1} = 1.4\times10^{-9}$$

We see, therefore, that requiring at least two intersecting collision pairs suffices to practically eliminate all false positives, while retaining a good portion of the true positives (same device). We can reduce false negatives by repeating the measurements with multiple mutually disjoint I sets.

Device tracking is achieved by calculating a set of loopback collisions from a measurement, and comparing (intersecting) it with all previous measurements to find a previous measurement with a sufficiently large intersection. If such a measurement is found, the current device is deemed to be identical to the device whose earlier measurement matched (and the current collision set can in fact be adjoined to it to improve the probability of future matches). If no matching earlier measurement is found, then this measurement is deemed to come from a new device, and a new entry is allocated to it in the list of measurements, with the current set of loopback intersections.