From IP ID to Device ID and KASLR Bypass

Amit Klein and Benny Pinkas, *Bar Ilan University*

https://www.usenix.org/conference/usenixsecurity19/presentation/klein

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

From IP ID to Device ID and KASLR Bypass[*]

Amit Klein
Bar-Ilan University

Benny Pinkas
Bar-Ilan University

Abstract

IP headers include a 16-bit ID field. Our work examines the generation of this field in Windows (versions 8 and higher), Linux and Android, and shows that the IP ID field enables remote servers to assign a unique ID to each device and thus be able to identify subsequent transmissions sent from that device. This identification works across all browsers and over network changes. In modern Linux and Android versions, this field leaks a kernel address, thus we also break KASLR.

Our work includes reverse-engineering of the Windows IP ID generation code, and a cryptanalysis of this code and of the Linux kernel IP ID generation code. It provides practical techniques to partially extract the key used by each of these algorithms, overcoming different implementation issues, and observing that this key can identify individual devices. We deployed a demo (for Windows) showing that key extraction and machine fingerprinting works in the wild, and tested it from networks around the world.

1 Introduction

Online browser-based user tracking is prevalent. Tracking is used to identify users and track them across many sessions and websites on the Internet. Tracking is often performed in order to personalize advertisements or for surveillance purposes. It can either be done by sites that are visited by users, or by third-party companies which track users across multiple web sites and applications. [2] specifically lists motivations for web-based fingerprinting as "fraud detection, protection against account hijacking, anti-bot and anti-scraping services, enterprise security management, protection against DDOS attacks, real-time targeted marketing, campaign measurement, reaching customers across devices, and limiting number of access to services".

Tracking methods Existing tracking mechanisms are usually based on either *tagging* or *fingerprinting*. With tagging, the tracking party stores at the user's device some information, such as a cookie, which can later be tracked. Modern web standards and norms, however, enable users to opt-out from tagging. Furthermore, tagging is often specific for one application or browser, and therefore a tag that was stored in one browser cannot be identified when the user is using a different browser on the same machine, or when the user uses the private browsing feature of the browser. Fingerprinting is implemented by having the tracking party measure features of the user's machine (for example the set of installed fonts). Corporates, however, often install a single "golden image" (standard set of software packages) on many *identical* (hardware-wise) machines, and therefore it is hard to obtain fingerprints that distinguish among such machines.

In this work we present a new tracking mechanism which is based on extracting data used by the IP ID generator (see Section 1.1). It is the first tracking technique that is able to simultaneously (a) cross the private browsing boundary (i.e. compute the same tracking ID for a private mode tab/window of a browser as for a regular tab/window of the browser); (b) work across different browsers; (c) address the "golden image" problem; and (d) work across multiple networks; all this while maintaining a very good coverage of the platforms involved. To our knowledge, no other tracking method (or a combination of several tracking techniques) achieves all these goals simultaneously. Moreover, the Windows variant of this technique also survives Windows shutdown+startup (but not restart).

Our techniques are realistic: for Windows we only need to have control over 8-30 IP addresses (in 3-13 class B networks), and for Linux/Android, we only need to control 300-400 IP addresses (can be in a single class B network). The Windows technique was successfully tested in the wild.

1.1 Introduction to IP ID

The IP ID field is a 16 bit IP header field, defined in RFC 791 [11]. It is used to facilitate de-fragmentation, by marking IP fragments that belong to the same IP datagram. The IP protocol assembles fragments into a datagram based on the fragment source IP, destination IP, protocol (e.g. TCP or UDP) and IP ID. Thus, it is desirable to ensure that given the same source address, destination address and protocol, the IP ID does not repeat itself in short time intervals. Simultaneously, the IP ID should not be predictable (across different destination IP addresses) since "[IP ID] predictability allows traffic analysis, idle scanning, and even packet injection in specific cases" [30].

Designing an IP ID generation algorithm that meets both requirements is not straightforward. Since IPv4 was standardized, several schemes have emerged:

- Global counter – This approach was used in the early IPv4 days due to its simplicity and its non-repetition

[*]An extended version of this paper can be found at http://www.securitygalore.com/site3/usenix2019.

period of 65536 global packets. However it is extremely predictable and thus insecure, hence abandoned.

- Counter/bucket based algorithms – This family of algorithms, suggested by RFC 7739 [7, Section 5.3], is the focus of our work. It uses a table of counters, and a hash function that maps a combination of a source IP address, destination IP address, key and sometimes other elements into an index of an entry in the table. IP ID is generated by choosing the counter pointed to by the hash function, possibly adding to it an offset (which may depend on the IP endpoints, key, etc.), and finally incrementing the counter. The non-repetition period in this family is 65536 global packets, and at the same time knowing IP ID values for one pair of source and destination IP addresses does not reveal anything about the IP IDs of pairs in other buckets.
- Searchable queue-based algorithm – This algorithm maintains a queue of the last several thousand IP IDs that were used. The algorithm draws random IDs until one is found that is not in the queue. Then this ID is used as the next IP ID, pushed to the queue, and the least-recently used value is popped from the queue. This algorithm ensures high unpredictability, and guarantees a non-repetition period as long as the queue.

Windows (version 8 and later) and Linux/Android implement variants of the counter-based algorithm. MacOS and iOS implement a searchable queue algorithm.

1.2 Introduction to KASLR

KASLR (Kernel Address Space Layout Randomization) is a security mechanism designed to defeat attack techniques such as ROP (Return-Oriented Programming [27]) that rely on the predictability of kernel code addresses. KASLR-enabled kernels randomize the kernel image load address during boot, so that kernel code addresses become unpredictable. While, e.g. in the Linux x64 kernel, the entropy of the load address is 9 bits, a brute force attack is deemed irrelevant since each failure usually ends in a system freeze ("kernel panic"). A typical KASLR bypass enables the attacker to obtain a kernel address (from which, addresses to useful kernel code gadgets can be calculated as offsets) without de-stabilizing the system.

1.3 Our Approach

The IP ID generation mechanisms in Windows and in Linux (UDP only) both compute the IP ID as a function of the source IP address, the destination IP address, and a key K which is generated when the source machine is restarted and is never changed afterwards. We run a cryptanalysis attack which analyzes the IP ID values that are sent by a device and extracts the key K. This key can then be used to identify the source device, because subsequent attacks will yield the same key value (until the device is restarted).

In more detail, IP ID generation in both systems maintains a table of counters and uses a hash function to choose which counter is used for each connection. It seems hard to deploy an attack based on the *value* of the counter, since each IP ID might depend on a different counter. Instead, our attack techniques rely on identifying and exploiting *collisions* which map two destination IP addresses to the same counter. This enables us to extract information about the key that caused the hash values to collide (Linux), or (in Windows) extract information about the offset of the IP ID from the counter. These values depend on K and therefore enable us to learn K and identify the machine.

Our approach does not rely on an a-priori knowledge of the counter values. Moreover, after we reconstruct K, we can reconstruct the current counter values (in full or in part) by sending traffic to specially chosen IP addresses, obtaining their IP ID values and with the knowledge of K, work back the counter values that were used to generate them.

Linux/Android KASLR bypass Support for network namespaces (part of container technology) was introduced in Linux kernel 4.1. With this change, the key K was extended to include 32 bits of a kernel address (the address of the `net` structure for the current namespace). Thus, reconstructing K also reveals 32 bits of a kernel address, which suffices to reconstruct the full address and be able to bypass KASLR.[1]

Conclusion In general, our work demonstrates that the usage of a non-cryptographic algorithm for the generation of attacker observable values such as IP ID, may be a security vulnerability even if the values themselves are not security-sensitive. This is due to an attacker's ability to extract the key used by the algorithm generating the values, and use this key to track or attack the system.

1.4 Advantages of our Technique

Tracking machines based on the key that is used for generating the IP ID has multiple advantages:

Browser Privacy Mode: Since our technique exploits the behavior of the IP packet generator, it is not affected if the browser runs in privacy mode.

Cross-Browser: Since our technique exploits the behavior of the IP packet generator, it yields the same device ID regardless of the browser used. It should be noted that browsers (like Tor browser) that relay transport protocols through other servers are not affected by our technique.

Network change: Tracking works across different networks since our technique uses bits of K as a device ID, and K does not depend on the device's IP address or network.

[1]Through our IP ID attack we were also able to achieve partial KASLR bypass, and a partial list of loaded drivers, with regards to Windows 10 RedStone 4. This attack was based on an additional initialization bug in Windows. However, that bug was repaired in the October 2018 security update and the corresponding KASLR bypass is not effective anymore.

The "Golden Image" Challenge: Since each device generates its own key K in a random fashion at O/S restart, even devices with identical software and hardware will most likely have different K values and thus different device IDs.

Not easily turned off: IP ID generation is built into the kernel, and cannot be modified or switched off by the user. Furthermore, the Windows attack can use simple HTTP traffic. The Linux/Android attack requires WebRTC which cannot be turned off for mobile Chrome and Firefox.

VPN resistant: The device ID remains the same when the device uses an IP-layer VPN.

Windows shutdown+startup vs. restart: The Fast Startup feature of Windows 8 and later,[2] which is enabled by default, saves the kernel to disk on shutdown, and reloads it from disk on system startup. Therefore, K is not re-initialized on startup, and keeps its pre-shutdown value. This means that the tracking technique for Windows survives system shutdown+startup. On restart, in contrast, the kernel is initialized from scratch, and a new value for K is generated, i.e. the old device ID is no longer in effect.

Scalability: Our technique can support billions of devices (Windows, Linux, newer Androids), as the device ID is random, and thus ID collisions are only expected due to the birthday paradox. Thus the probability of a single device not to have a unique ID is very low.

It should be noted that in the Linux/Android case, due to the use of 300-400 IP addresses, the need to "dwell" on the page for 8-9 seconds, and (in newer Android devices) the excessive attack time, there are use cases in which the technique may be considered invasive and/or inapplicable.

Additional Contributions: In addition to the cross-browser tracking technique for Windows and Linux, and the KASLR bypass with respect to Linux, we also provide the first full public documentation of the IP ID generation algorithm in Windows 8 and later versions, obtained via reverse-engineering of the relevant parts of Windows kernel `tcpip.sys` driver, and a cryptanalysis of said algorithm. We also show a demo implementation of the Windows tracking technique and provide results from an extensive in-the-wild experiment spanning 75 networks in 18 countries, demonstrating the applicability of the attack.

We disclosed the vulnerabilities to Microsoft and Linux. Microsoft fixed the issue in Windows April 2019 Security Update (CVE-2019-0688).[3] Linux fixed the kernel address disclosure (CVE-2019-10639) together with partially addressing the key-based tracking technique (by extending the key to 64 bits) in a patch[4] applied to Linux kernel versions 5.1-rc4, 5.0.8, 4.19.35, 4.14.112, 4.9.169 and 4.4.179. For 3.18.139 and 3.16.67, Linux applied a patch[5] we developed, that extends the key to 64 bits. The key-based tracking technique (CVE-2019-10638) is fully addressed in a patch,[6] part of kernel version 5.2-rc1, and will be back-ported to kernel versions 5.1.7, 5.0.21, 4.19.48 and 4.14.124.

Note: many non-essential details of the attack, as well as proofs for false positive bounds for Windows, are deferred to the extended version of the paper.

2 The Setting

We assume that device tracking is carried out over the web, using an HTML snippet (which can be embedded by a 3rd party site/page). The snippet forces the browser to send TCP or UDP traffic (one packet per destination IP suffices) to multiple IP addresses under the tracker's control (8-30 addresses for Windows, 300-400 for Linux/Android). Ideally, such transmission would be rapid. In our experiments, this can be done in few seconds or less.

For the Windows attack, the tracker needs to choose the IP addresses according to some trivial constraints (the Linux IP addresses are not subject to any constraints). A discussion of the exact constraints and their trade-offs can be found in the extended paper. At the server side, the tracker collects the IP ID values sent by the client to each of the IPs, and computes a device ID consisting of bits of the key in the device's kernel data that is used to calculate the IP ID.

Additional scenarios (KASLR bypass and internal IP disclosure) for Linux/Android attacks are described in the extended paper.

3 Related Work

Many tracking techniques were suggested in prior research. At large, proposals can be categorized by their passive/active nature. We use the terminology defined in [31]:

- A *fingerprinting* technique measures properties already existing in the browser or operating system, collecting a combination of data that ideally uniquely identifies the browser/device without altering its state.
- A *tagging* technique, in contrast, stores data in the browser/device, which uniquely identifies it. Further access to the browser can "read" the data and identify the device.

[2]https://blogs.msdn.microsoft.com/olivnie/2012/12/14/windows-8-fast-boot

[3]https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-0688

[4]"netns: provide pure entropy for net_hash_mix()" (https://github.com/torvalds/linux/commit/355b98553789b646ed97ad801a619ff898471b92)

[5]"inet: update the IP ID generation algorithm to higher standards" (https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=55f0fc7a02de8f12757f4937143d8d5091b2e40b)

[6]"inet: switch IP ID generator to siphash" (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=df453700e8d81b1bdafdf684365ee2b9431fb702)

As described in Section 1, fingerprinting techniques typically cannot guarantee the uniqueness of the device ID, in particular with respect to corporate machines cloned from “golden images”. Tagging techniques store data on the device, and as such they are more easily monitored and evaded. A comprehensive discussion of tracking methods can be found in Google Chromium’s web page “Technical analysis of client identification mechanisms” [12].

3.1 IP ID Research

Device tracking via IP ID: Using IP ID is proposed in [5] (2002) to detect multiple devices behind a NAT, assuming an IP ID implementation using a *global* counter. But nowadays none of the modern operating systems implements IP ID as a global counter. A similar concept is presented by [25] for a single destination IP (the DNS resolver) which theoretically works for devices that have per-IP counter (Windows, to some extent). However, this technique does not scale beyond a few dozen devices, due to IP ID collisions (the IP ID field provides at most 2^{16} values), and requires ongoing access to the traffic arriving at the DNS resolver.
Predictable IP ID: The predictability of IP ID may theoretically be used in some conditions to track devices. [6] describes a technique to predict the IP ID of a target, but requires the adversary to have a fully controlled device alongside it behind the same NAT. Also this technique only handles sequential increments (e.g. not time-based). As such, it is inapplicable to the more general scenarios handled in this paper. This technique is then used in [9] to poison DNS records.
OS Fingerprinting: [32] suggests using $IPID = 0$ as a fingerprint for some operating systems.
Measuring traffic: [29] samples IP ID values from servers whose IP ID is a global counter, to estimate their outbound traffic.
IP ID Algorithm Categorization: [28] provides practical classification of IP ID generation algorithms and measurements in the wild.
Fragmentation attacks: While not directly related to the properties of the IP ID field, it should be noted that attack techniques abusing fragmentation are known. RFC 1858 [26] lists several such attacks, e.g. the “tiny frgament” attack and the “overlapping fragement” attack.

Windows IP ID research: In parallel to our research, Ran Menscher published on Twitter his research on Windows IP ID [23]. That research reverse-engineered part of the Windows IP ID generation algorithm (without revealing how the index to the counter array is calculated). The analysis of this algorithm is based on two assumptions: (1) that the technique is applied shortly after restart, when the relevant memory buffer contains zeroes in a large part of its cells; and (2) that the attacker controls or monitors traffic to pairs of IP addresses which differ in single, specific bit position (including positions in the left half of the address). Based on these extreme assumptions, the attacker can extract the key easily, and use it to expose kernel 31-bit data quantities (though without learning where in the array this data resides).

The uninitialized memory issue exploited by this attack was fixed in Microsoft’s October 2018 Security Update [24], which invalidated assumption (1), rendering Menscher’s attack completely ineffective. Our attack and our demo, on the other hand, still work against systems that were patched with this update. Our work has multiple contributions over Menscher’s attack: (1) We provide the full details of the IP ID algorithm. (2) Our analysis does not rely on the array data, and is thus still in effect after applying the October 2018 Security Update which initializes the array with random data. (3) Our analysis does not require the extreme requirements on the relations between the addresses of the controlled/monitored IP addresses. (4) Our kernel data exposure provides positions of the data, not just data quantities (though our kernel exposure technique, too, was eliminated with the October 2018 Security Update). (5) It should also be noted that unlike our attack, Menscher’s technique could not be used for tracking, since as the cell arrays become non-zero when they are incremented, the attack becomes ineffective.

3.2 PRNG seed/key extraction

Our approach involves breaking the random number generator algorithm used by operating systems to generate the IP ID value and obtain the seed/key used by the algorithm. Similar strategies were used to different ends. For example, [17] broke the PRNG of the Witty worm to obtain the seed, from which they learned the infection time of the Internet nodes. [14] broke the Javascript `Math.Random()` PRNG of several browsers, obtained the seed and used it as a browser instance tracking ID. [15] broke the `Math.Random()` PRNG of Adobe Flash, obtained the seed and used it to extract the machine clock speed.

4 Tracking Windows 8 (and Later) Devices

In this section we first present the algorithm that is used for generating the IP ID in Windows 8 (and later) devices. The input to this algorithm includes a key which is generated at system restart. We then describe how a remote server can identify 45 bits of this key. This data enables to remotely and uniquely identify machines.

4.1 IP ID Generation

IP ID prior to Windows 8 In versions of Microsoft Windows up to and including Windows 7, the IP ID was generated sequentially and globally. That is, for each outgoing IP packet, a global counter would be incremented by 1 and the result (truncated to 16 bits) would be used [25]. These older Windows versions are out of scope for this paper.

The source code of the algorithm that is used for generating IP ID values in Windows is not public. However, we recovered the exact algorithm using reverse engineering, and verified its correctness by comparing its output to IP ID values generated by live Windows systems.

Technical details The algorithm was obtained by reverse-engineering parts of the `tcpip.sys` driver of 64-bit Windows 10 RedStone 4 (April 2018 Update, Build 1803). Apparently this algorithm is in use starting with Windows 8 and Windows Server 2012. Notice that the code is not specific to IPv4, and can be used with IPv6, which is why the key K is defined as 320 bits - more than required to support IPv4.[7] For IPv4 pre RedStone 5, only 106 key bits are used.

Toeplitz hash The IP ID generation is based on the Toeplitz hash function defined in [10]. Let us first define the *Toeplitz hash*, $T(K,I)$, which is a bilinear transformation from a binary vector K in $GF(2)^{320}$, and an input which is a binary string I (where $|I| \leq 289$) to the output space $GF(2)^{32}$. For a binary vector V, denote by V_i the i-th bit in the vector, with bit numbering starting from 0. The i-th bit of $T(K,I)$ ($0 \leq i \leq 31$) is defined as the inner product between I and a substring of K starting in location i. Namely

$$T(K,I)_i = \bigoplus_{j=0}^{|I|-1} I_j \cdot K_{i+j} \tag{1}$$

IP ID generation The IP ID generation algorithm itself uses keys K (`tcpip!TcpToeplitzHashKey`) which is a 320 bit vector, and $K1$ and $K2$ which are 32 bits each. All these keys are generated once during Windows kernel initialization (using `SystemPrng` and `BCryptGenRandom`).

In addition to these constant keys, the algorithm uses a dynamic array of M counters, denoted $\beta[0],\ldots,\beta[M-1]$, where M is a power of 2, and is specifically set to $M = 8192$.

Algorithm 1 describes how Windows 8 (and later) generates an IP ID for a packet delivered from IP_{SRC} to IP_{DST}, while updating a counter in β.

The algorithm uses the keys, and the source and destination IP addresses, to pick a random index i for a counter in β, and an offset. The algorithm outputs the sum of the counter $\beta[i]$ and the offset, and increments the counter.

Notation We use the notation $\mathrm{Num}(a_0,a_1,\ldots,a_{31})$ for the number represented in binary by the bits a_i, namely the number $\sum_{i=0}^{31} a_i \cdot 2^{31-i}$. (Network byte order is used throughout the paper for representing IP addresses as bit vectors, e.g. 127.0.0.1 is 01111111.00000000.00000000.00000001.)

Properties of the Toeplitz hash Our attack uses the following properties of T, which follow from the linearity of this transformation:

$$T(K,I||(0,0,\ldots,0)) = T(K,I) \tag{2}$$

[7] Our tracking technique can be probably adapted to IPv6, but since IPv6 is out of scope for this paper, we did not test this.

Therefore the trailing zeros in the input of T in the computation of v on line 3 of Algorithm 1, have no effect on the output. Also,

$$T(K,I_1||I_2) = T(K,I_1) \oplus T(K,0^{|I_1|}||I_2) \tag{3}$$

Therefore it is possible to decompose the second input of T to two parts, and rephrase the computation as the XOR of two separate expressions.

4.2 Reconstructing the Key K

To reconstruct the key, the device needs to be measured. The measurements only take a few seconds, and are thus assumed to take place from the same network. I.e., the *device's* source IP address, IP_{SRC}, is fixed (though possibly unknown). A first set of measurements directs the client device to J IP addresses from the same class B network. A second set of measurements directs the client device to G *pairs* of IP addresses, each pair in the same class B network, with G different class B network pairs in the set.

Once the device is measured, the attack proceeds in two phases. The first phase of the attack recovers 30 bits of the key using the first set of measurements. The second phase of the attack reveals additional 15 bits of the key using the second set of measurements. Overall, the measurements reveal 45 bits of the key, which suffices to uniquely identify machines from a large population, with high probability.

Section 4.5 describes how to optimally choose the parameters J and G given limits on the number of IP addresses that are available (L) and the processing time that is allowed ($\mathcal{T}$). For $L = 30$ IP addresses (typical low budget limit), and attack run time limit of $\mathcal{T} = 1$ seconds on a single Azure B1s machine ($\alpha = 0.001$ from Section 5.2), the optimal parameter values are $J = 6, G = 12$.

4.3 Extracting Bits of K - Phase 1

Denote by $IP^{g,j}$, $IPID^{g,j}$ and $\beta[i_g]^{g,j}$ the values of the destination IP address, the IP ID and $\beta[i]$ (prior to increment) respectively, with respect to the j-th packet in the g-th class B network that is used in the attack (j and g are counted 0-based). The first phase of the attack uses only a single class B network, and therefore g is set to 0 in this phase. We thus use the following shorthand notation: $IP^j = IP^{0,j}$, $IPID^j = IPID^{0,j}$ and $\beta_g = \beta[i_g]^{g,0}$.

A major observation is that only the first half of IP_{DST} is used to calculate i in Algorithm 1. Therefore packets that are sent to different IP addresses in the same class B network, have an identical index i into the counter table, and use the same counter $\beta[i]$. Denote the value of i for the g-th class B network as i_g.

If these packets are sent in rapid succession (i.e. when no other packet is sent in-between with $i = i_g$), then $\beta[i_g]^{g,j} = \beta_g + j \mod 2^{32}$, and therefore the output in line 5 of the

Algorithm 1 Windows 8 (and later) IP ID Generation

```
1: procedure GENERATE-IPID
2:     $i \leftarrow \mathrm{Num}(K2 \oplus T(K, (IP_{DST})_{0,\ldots,\frac{|IP_{DST}|}{2}-1}) \oplus T(K, IP_{SRC})) \mod M$
3:     $v \leftarrow \beta[i] + \mathrm{Num}(K1 \oplus T(K, IP_{DST}||IP_{SRC}||0^{32})) \mod 2^{32}$
4:     $\beta[i] \leftarrow (\beta[i]+1) \mod 2^{32}$
5:     return $v \mod 2^{15}$                    $\triangleright$ $v \mod 2^{16}$ for Windows 10 RedStone 5
```

algorithm is calculated with $\beta[i_g]^{g,j} = \beta_g + j \mod 2^{15}$ (for simplicity, in Windows 10 RedStone 5, we discard the most significant bit of the IP ID).

We focus in this phase on the first class B network, b_0, with J destination IP addresses in it. Note that the offset that is calculated in line 3 is the difference between the IPID and the counter $\beta[i_0]$ prior to its increment.

The attack enumerates over the values of the $\beta_0 \mod 2^{15}$ counter. For each possible value it calculates the differences between the observed J IPIDs and the corresponding values of the counter, arriving at the offsets calculated in line 3. By observing pairs of IPIDs, it is possible to identify the correct value of $\beta_0 \mod 2^{15}$ as well as 30 bits of the key.

In more detail, for each possible value of $\beta_0 \mod 2^{15}$ the attack calculates the difference

$$IPID^j - (\beta_0 + j \mod 2^{15}) \mod 2^{15}$$

which, for the right value of the counter should be equal to the offset that is calculated in line 3. Namely to

$$\mathrm{Num}(K1 \oplus T(K, IP^j||IP_{SRC}||0^{32})) \mod 2^{15}$$

This value can be expressed as $(K1 \oplus T(K, IP^j||IP_{SRC}||0^{32}))_{17,\ldots,31}$. Applying eq. (2) and eq. (3), this expression is simplified into:

$$(K1 \oplus T(K, IP^j) \oplus T(K, 0^{32}||IP_{SRC}))_{17,\ldots,31}$$

.

The attack takes two different j values and computes the XOR of the two corresponding such quantities. This results in the following expression (where we denote by Vec a representation of a number in $[0, 2^{32})$ as a vector in $GF(2)^{32}$):

$$(\mathrm{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus \mathrm{Vec}(IPID^{j'} - (\beta_0 + j') \mod 2^{15}))_{17,\ldots,31} = T(K, IP^j \oplus IP^{j'})_{17,\ldots,31}$$

This yields 15 linear equations ($i = 17,\ldots,31$) on K since (from eq. (1)):

$$T(K, IP^j \oplus IP^{j'})_i = \bigoplus_{m=0}^{31} (IP^j \oplus IP^{j'})_m \cdot K_{i+m}$$

Since all IP^j belong to the same class B network, $IP^j \oplus IP^{j'}$ always has 0 for its first 16 bits, and therefore m can start at 16. Due to obvious linear dependencies, only $J-1$ sets of such equations are useful (e.g. all pairs with $j' = 0$), with a total of $15(J-1)$ linear equations for bits $K_{33},\ldots,K_{62}$. That is, for $j = 1,\ldots,J-1$ and $i = 17,\ldots,31$, the equations are:

$$\bigoplus_{m=16}^{31} (IP^j \oplus IP^0)_m \cdot K_{i+m} = (\mathrm{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus \mathrm{Vec}(IPID^0 - (\beta_0) \mod 2^{15}))_i \quad (4)$$

Speeding up the computation using preprocessing The coefficients of K in eq. (4) are controlled by the server and are known at setup time. Therefore it is possible to preprocess the computation of Gaussian elimination. Namely, compute a matrix Z that, when multiplied by the observed values, reveals bits of the key. This preprocessing is only important for efficiency, therefore we defer the details to the extended paper.

Attack summary

1. The tracker needs to control J IP addresses in the same class B network.
2. During setup time, the tracker calculates, using Gaussian elimination, a matrix $Z \in GF(2)^{15(J-1)\times 15(J-1)}$, based on the values of these IP addresses.
3. In real time, the tracker gets IP ID values from the device, from packets sent to the J destination IP addresses under the tracker's control.
4. The tracker then guesses 14 bits ($\beta_0 \mod 2^{14}$ - the most significant bit of $\beta_0 \mod 2^{15}$ cancels itself in eq. (4)) of the counter that is used for these IP addresses, calculates vectors D^j ($j = 1,\ldots,J-1$), where $D^j = (\mathrm{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus \mathrm{Vec}(IPID^0 - (\beta_0) \mod 2^{15}))_{17,\ldots,31}$, and performs a matrix-by-vector multiplication of Z and the vector $(D^0,\ldots,D^{J-1})$.

   For the correct value of $\beta_0 \mod 2^{14}$ this computation results in a vector of $15(J-1)$ bits, whose first 30 bits are $K_{33},\ldots,K_{62}$ and the remaining bits are zero.
5. The attacker identifies the right value of the counter by comparing to zero the $15(J-1) - 30$ bits starting at position 31: if $15(J-1) - 30 \gg 14$, this verification statistically guarantees the correctness of the solution (up to a flipped most significant bit in $\beta_0 \mod 2^{14}$, see the extended paper.)

Overall this process reveals 30 bits of the key as well as the value ($\beta_0 \mod 2^{14}$).

The attack takes $2^{14} \cdot (15(J-1))^2$ bit operations (for enumeration over the possible key values and for the matrix-by-vector and $(15(J-1))^2$ memory bits (for Z). As explained in Section 4.5, we set $J = 6$ and therefore this overhead is very small.

The tracker obtains the (correct) value $\beta_0 \mod 2^{14}$, which will be used in the next phase. While it is guaranteed that the correct K and $\beta_0 \mod 2^{14}$ will be found, the algorithm may emit additional candidates (with incorrect $\beta_0 \mod 2^{14}$). The false positive probability of both phases of the attack is analyzed in the extended paper.[8]

4.4 Extracting Bits of K - Phase 2

Given 30 bits of K ($K_{33}, \ldots, K_{62}$) and the value ($\beta_0 \mod 2^{14}$), recovered in Phase 1, the attack can be extended to learn a total of up to 45 key bits ($K_{18}, \ldots, K_{62}$). This is done in the following way. The offset for $IPID^0$ computed in line 3 of Algorithm 1 is:

$$\text{Num}(K1 \oplus T(K, IP^0) \oplus T(K, 0^{32}||IP_{SRC})) \mod 2^{15} =$$
$$(IPID^0 - \beta_0) \mod 2^{15}$$

The following equation follows from the previous one:

$$(K1 \oplus T(K, 0^{32}||IP_{SRC}))_{17,\ldots,31} = T(K, IP^0)_{17,\ldots,31} \oplus$$
$$\text{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}$$

The tracker looks at pairs of IP addresses in the remaining B classes ($b_1, \ldots, b_G$), each pair in a different class B network. Denote each such pair as ($IP^{g,0}, IP^{g,1}$), with the order inside the pair conforming to the order of packet transmission, and the packets being transmitted in rapid succession. Substituting the above into the definition of $IPID$ yields:

$$\begin{aligned} IPID^{g,j} &= \beta_g + j + \text{Num}(\ T(K, IP^0)_{17,\ldots,31} \\ &\oplus \text{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31} \\ &\oplus T(K, IP^{g,j})_{17,\ldots,31}\ ) \mod 2^{15} \end{aligned}$$

Using the linearity of T, this is simplified into:

$$IPID^{g,j} = \beta_g + j + \text{Num}\ (\ T(K, IP^0 \oplus IP^{g,j})_{17,\ldots,31} \oplus$$
$$\text{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}\ ) \mod 2^{15}$$

Let us use the notation

$$S^{g,j} = \text{Num}\ (\ T(K, IP^0 \oplus IP^{g,j})_{17,\ldots,31} \oplus$$
$$\text{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}\ ) \mod 2^{15}$$

[8]Note: Throughout the paper, we assume that $\text{rank}(C) = 30$. This results in a single key vector per guessed $\beta_0 \mod 2^{14}$. We discuss the conditions on $IP^0, \ldots, IP^{J-1}$ to meet this assumption in the extended paper. If $\text{rank}(\ker(C)) > 0$, then each guess of $\beta_0 \mod 2^{14}$ yields $2^{\text{rank}(\ker(C))}$ possible keys. Thus small values of $\text{rank}(\ker(C))$ are acceptable.

Then this equation becomes

$$IPID^{g,j} = \beta_g + j + S^{g,j} \mod 2^{15}$$

Subtracting the IPIDs of the two consecutive packets in the same B class (with $j = 0$ and $j = 1$) cancels the value of the counter β_g, and yields:

$$(IPID^{g,1} - IPID^{g,0}) \mod 2^{15} = 1 + S^{g,1} - S^{g,0} \mod 2^{15} \tag{5}$$

The left side of the equation is observed by the tracker. The right side can be computed based on $\beta_0 \mod 2^{15}$ and $K_{17}, \ldots, K_{62}$. The tracker already knows these values except for $K_{18}, \ldots, K_{33}$, and therefore only needs to enumerate over the 2^{15} possible values of $K_{18}, \ldots, K_{32}$ and eliminate all values which do not agree with the equation. We discuss this procedure in depth in the extended paper.

Attack summary:

1. The tracker needs to control additional G pairs of IPs (each pair in its own class B network).
2. Given IP IDs for these pairs, the tracker enumerates over additional 15 key bits, and then, for each pair of IP addresses, calculates both sides of eq. (5) and compares them. For this calculation the tracker can choose K_{17} and the leftmost bit of $\beta_0 \mod 2^{15}$ arbitrarily, as they will both cancel themselves.
3. In theory, each IP pair should yield a 2^{15} elimination power for identifying the right key, but see the extended paper for a more accurate analysis.
4. In the calculation, the leading term (in terms of run time) is computing $T(K, I)_{17,\ldots,31}$ (where $|I| = 32$), which takes $14|I|$ bit operations, and is used twice. Thus, the run-time is roughly $2^{15} \cdot 2 \cdot 14 \cdot 32$ bit operations (there is no multiplication by G since the first pair is likely to eliminate almost all false guesses).

At the end of Phase 2, the tracker obtains:

- A partial key vector (or some candidates) $K_{18}, \ldots, K_{62}$ (45 bits), which is specific to the device since it was set during kernel initialization, and does not depend on IP_{SRC}. These bits serve as a device ID.
- The value

$$(K1 \oplus T(K, 0^{32}||IP_{SRC}))_{18,\ldots,31} = T(K, IP^0)_{18,\ldots,31}$$
$$\oplus \text{Vec}(IPID^0 - \beta_0 \mod 2^{14})_{18,\ldots,31}$$

  This value allows the tracker to calculate (assuming $K_{18}, \ldots K_{62}$ are known) the value of the counter $\beta[i] \mod 2^{14}$ for any destination IP address whose IP ID is known (provided the source IP is IP_{SRC}).[9]

[9]This is useful for reconstructing the table β of counters – this table is not correctly initialized (pre October 2018 Security Update), and therefore is populated with kernel data that happens to be (in build 1803) data structures containing kernel address pointers.

4.5 Choosing Optimal G and J

For Windows, we assume budget-oriented constraints, namely L available IP addresses and $\mathscr{T}$ CPU time per measurement. We need to set the number J of IP addresses from the same class B network to which the client is directed in the first set of measurements, and the number G of pairs of IP addresses, each pair in the same class B network, used in the second set of measurements.

Our goal is to optimize for minimum false positives. The first constraint can be expressed as $J+2G \leq L$. As for the second constraint, the leading term of the time of the attack run is $\alpha \cdot (J!)$ (Appendix A.1.2), where α expresses the computing platform's strength. Therefore, we can approximate the second constraint as $\alpha \cdot (J!) \leq \mathscr{T}$. Additionally, there are inherent constraints: $J-1 \geq 3$ to let Phase 1 suggest a single key candidate to Phase 2 (most of the time), and $G \geq 2$ to let Phase 2 provide a single final key (most of the time).

Given these constraints, we want to minimize the leading term in false positives, $2 \cdot 2^{-\frac{G+J-1}{2}}$ (Appendix A.2), i.e. we need to maximize $G+J$. Since we "pay" two IP addresses for each increment of G and only one IP address for each increment of J, we should make J as large as possible (as long as G is valid), so the solution is:

$$J = \min(\max(\{J \mid \alpha J! \leq \mathscr{T}\}), L-4)$$

(As stated in Section 4.2, for $L=30$, $\mathscr{T}=1$ sec., and $\alpha = 0.001$, the optimal combination is $J=6, G=12$.)

4.6 Practical Considerations

We discuss in Appendix A.1 different issues that appear when deploying the attack. These issues include ways to emit the needed traffic from the browser, handling packet loss and out-of-order packet transmission, handling interfering packets, and limiting the false-positive and false-negative error probabilities.

The run time of the key extraction attack is less than a second even on a very modest machine. The *dwell time* (time duration in which the page needs to be loaded in the browser) is 1-2 seconds for a WebSocket implementation. It is possible to minimize the dwell time by moving to WebRTC (STUN).

Longevity: the device ID is valid until the machine restarts (mere shutdown+start does not invalidate the device ID due to Windows' Fast Start feature). A typical user needs to restart his/her Windows machine only for some Windows updates, i.e. with a frequency of less than once per month.

The attack is scalable: with 41 bits, the probability of a device to have a unique ID is very high, even for a billion device population; false positives are also rare (2.1×10^{-6} – Table 3), and false negatives can be made negligible (Appendix A.1.4). From resource perspective, the attack uses a fixed number of servers, RAM/disk and ($L=30$) IPs. The required CPU power is linear in the number of devices measured per time unit, and in the Windows case is negligible. Network consumption per test is also negligible (assuming WebRTC/STUN implementation – 1.5KB at the IP layer.)

4.7 Attack Improvements and Variants

A **fast-track identification of already-seen keys** can be obtained in the following way: Once bits of a key K are extracted, they will be stored for comparison against future connections. When a device is to be measured, the tracker first goes through all stored K bit strings, and tests the measured data for compatibility with each one of them. This amounts to guessing the bits of β_0 one by one, starting from the least significant, and eliminating via eq. (5), using mod 2^n where n is the number of β_0 bits guessed so far. The CPU work per key is thus almost negligible.

The original attack can also be sped up using **incremental evaluation**. The details are in the extended paper.

4.8 Environment Factors

We demonstrate here that the tracking attack can be deployed in almost every setting that can be reasonably expected.

HTTPS: In essence, there should be no problem in having the snippet use WebSocket over HTTPS (`wss://` URL scheme) for TCP packets.

NAT: Typically NAT (Network Address Translator) devices do not alter IP IDs, and thus do not affect the attack.

Transparent HTTP Proxy / Web Gateway: Such devices may terminate the TCP connection and establish their own connections (with IP ID from their own network stack) and thus render our technique completely ineffective. However, typically these devices do not interfere with HTTPS (TCP port 443) traffic, and UDP traffic, so these alternatives can be used by the tracker.

Forward HTTP proxy: When a browser is configured to use a forward proxy server, even HTTPS traffic is routed to it by the browser. However, it may still be the case that UDP traffic (which is not handled by HTTP forward proxies) can be used by the technique.

Tor-based browsers and similar browsers: Browsers that forward TCP traffic to proxy servers (and disallow or forward UDP requests) are incompatible with the tracking technique as they do not expose IP header data generated on the device. Since "Tor transports TCP streams, not IP packets",[10] this applies to all Tor-based products, such as the Tor browser and Brave's "Private Tabs with Tor" and therefore they are not covered by our technique.

Windows Defender Application Guard (WDAG): This new technology in Windows 10 enables the user to launch the Edge browser in a virtual environment. While the device ID

[10] `https://www.torproject.org/docs/faq.html.en#RemotePhysicalDeviceFingerprinting`

in this virtual environment is independent of the device ID of the main operating system, it is consistent among **all** WDAG Edge instances. Furthermore, unlike the "main" Windows device ID, the WDAG device ID does not change with operating system restart, hence the WDAG device ID lives longer than the main Windows device ID. It should be noted that WDAG is only available for Edge browser in Windows 10 Enterprise/Pro edition, and requires high-end hardware.

IP-Level VPN: We experimented with F-Secure Free-Dome (`www.f-secure.com/en/web/home_global/freedome`) and PureVPN (`www.purevpn.com/`). Both VPNs supported our technique.

IPv6 and IPsec: We do not know whether IPv6 or IPsec packets use the same IP ID generation mechanism. This requires further research.

Javascript disabled: Tracking can also work when Javascript (or any client side scripting) is not available, e.g. with the NoScript browser extension [20]. We discuss this in the extended paper.

4.9 Possible Countermeasures

We list here some obvious ways of modifying Algorithm 1 and their impact:

- Increasing M (the size of the table of counters) – surprisingly, this has very little effect on the basic tracking technique, since no assumptions were made on M in the first place. It does affect the β reconstruction technique.
- Changing T into a cryptographically strong keyed-hash function – while this change eliminates the original attack, it is still possible to mount a weaker attack that only tracks a device while its IP_{SRC} does not change. In fact, this applies to the entire abstract scheme proposed in [7, Section 5.3]. See the extended paper for details.
- Changing the algorithm altogether (this is our recommendation). A robust algorithm relies on industrial-strength cryptography, large enough key space, and strong entropy source for the key, and uses them to generate IP IDs which (a) have guaranteed non-repetition period; (b) are difficult to predict; and (c) do not leak useful data. The algorithm used in macOS/iOS [30] is a good example. This eliminates the attack altogether.

5 Field Experiment – Attacking Windows Machines in the Wild

We set up a fully operational system to test the IP ID behavior in the wild, as well as to verify that the technique for extracting device IDs for Windows machine works as expected.

5.1 Setup

As explained in Appendix A.1.3, in order to avoid false positives (which almost always happen due to false keys that differ from the true key in a few most significant bits), we need to trim the most significant bits from the key – i.e. use the key's tail. For the full production setup (30 IP addresses), we calculated that a tail of 41 bits will suffice. Due to logistic and budgetary constraints, in our experiment we used only 15 IP addresses (rather than 30) for the key extraction (and 2 more IPs for verification), with $J = 5, G = 5, Q = 1$. Thus we lowered the tail length to 40, and used the 40 bits $K_{23}, \ldots, K_{62}$ as a device ID. That is, for this experiment, we traded the device ID space size for a smaller probability of false positives.

We then used WebSocket traffic to the additional pair of IP addresses (from a class B network that is different than those in the initial set of 15 IPs) to verify the correctness of the key bits extracted. In this experiment, since we do not extract $K_{17,\ldots,22}$ we can only compute the least significant 9 bits of the IPID, adapting eq. (5) into:

$$IPID^{g,1} \mod 2^9 = IPID^{g,0} \pm 1 + S^{g,1} - S^{g,0} \mod 2^9$$

(We need to use ± 1 since we cannot know the order of packet generation. Thus given knowledge of $IPID^{g,0}$ we have two candidates for $IPID^{g,1}$, out of a space of $2^9 = 512$ values.) A random choice of two values yields a success rate of $1/256$. We deem our algorithm to be valid if it consistently yields the correct value (in one of the candidates) in all tests.

We asked "Friends and Family" to browse to the demo site using Windows 8 or later, from various networks.

5.2 Results

Network distribution The experiment was conducted from July 22nd, 2018 to October 20th, 2018. We collected data on 75 different class B networks. The networks are well dispersed across 18 countries and 4 continents. The networks are also usage-diverse (home networks, SMB networks, corporate networks, university networks, public hotspots and cellular networks). We asked the users who connected to our demo site to use multiple regular browsers and networks, and connect at different times, and verified that the device ID remained the same in all these connections.

Failures to extract a key – IP ID modification In only 6 networks out of 75 (8%) we could not extract the key and therefore concluded that the IP ID was not preserved by the network. These six networks did not include any major ISP and seem to be used by relatively few users: they included an airport WiFi network, a government office, and a Windows machine connecting through one cellular hotspot (hotspots that we tested in other cellular networks did not change the IP ID). Of those six networks, in 3 networks we had clear indication that a transparent proxy or a web security gateway was in path. In such cases, moving to WebSocket over HTTPS, or to UDP would probably have addressed the issue. Another case was a forward proxy (moving to UDP would have possibly addressed it). In the two final cases, the exact nature of interference was not identified. We can say then that optimistically, only 2 networks out of 75 (2.7%) are incompatible with the tracking technique, maybe even less (as it is still quite possible these two TCP gateways are actually transparent proxies).

Positive results In the remaining 69 networks, for 4 networks we did not keep traffic for the additional two IPs, thus we could not verify the key extraction. For the rest 65 networks, our algorithm extracted a single 40-bit key, and correctly predicted the least significant 9 bits of the IPID of the second IP in the last pair (i.e. the correct value was one of the two candidates computed by the algorithm). This verifies the correctness of the algorithm and the key bits it extracts.

Lab verification We tested a machine in the lab with the above test setup to obtain 40 bits of K. Then, using WinDbg in local kernel mode, we obtained `tcpip!TcpToeplitzHashKey`, extracted the 40 bits from it and compared to the 40 bits calculated by the snippet – as expected, they came out identical.

Actual run time We estimate the overall runtime for $J = 6, G = 12$ on a single Azure B1s machine to be 0.73 seconds.

Packet loss and false negatives We analyzed 79 valid tests and found only 3 cases wherein the analysis logic failed to provide a device ID (additional test from the same devices succeeded in extracting a key). In all such cases a manual analysis indicates that this is due to packet loss. Appendix A.1.4 describes additional logic that can be used to reduce false negatives to a negligible level.

6 Linux and Android

The scope of our research is Linux kernel 3.0 and above. Also, we only investigated the x64 (typical desktop Linux) and ARM64 (Android) CPU architectures, although almost all of the analysis is not architecture-specific.

6.1 Attack Outline

In order to track a Linux/Android device, the tracker needs to control several hundred IP addresses. The tracking snippet forces the browser to rapidly emit UDP packets to each such IP (using WebRTC and specifically the STUN protocol, which enables sending bursts of packets closely spaced in time to controlled destination addresses). It also collects the device's source IP address (using WebRTC as well or a different approach described in the extended paper.)

The tracker collects IP IDs from all IP addresses, and identifies bucket collisions by looking for IP pairs whose IP IDs are in close proximity. Recall that the choice of the bucket is a function of the source and destination IP addresses, and a device key. The tracker enumerates over the key space to find the (correct) key which generates collisions for the same pairs for which collisions were observed. The key that is found is the device ID.

6.2 IP ID Generation in Linux

The Linux kernel implementation of IP ID differs between TCP and UDP [16]. The TCP implementation always used a counter per TCP connection (initialized with a hash of the connection endpoints and a secret key, combined with a high resolution timer) and as such, is not interesting to us (collisions are meaningless). The implementation of IP ID for stateless over-IP protocols (e.g. UDP) has gone through an interesting evolution process. We focus on short datagrams, i.e. datagrams shorter than MTU (maximum transmission unit), that do not undergo fragmentation. We designate the IP ID generation algorithms as A_0, A_1, A_2 and A_3, in their order of evolution.

A_0**:** In early Linux kernels, the IP ID for short datagrams was simply set to 0.

A_1 **and** A_2**:** In Linux kernel 3.16.0 (released August 2014), IP ID for short datagrams became dynamic (just like it has always been for long UDP datagrams).[11] This was back-ported to various active Linux 3.x branches (see Table 2). The generation algorithm in general has an array of $M = 2048$ buckets, each containing a value $0 \leq \beta < 2^{16}$ and a time-stamp τ of the last time this bucket was used. The bucket array is initialized at boot time with random data (using a PRNG). The algorithm also uses the following parameters

- *key* – a 32-bit key (`ip_idents_hashrnd`) which is initialized upon first IP transmission with random data.
- h – a hash function. Older and newer versions of Linux used different hash functions (A_1 and A_2, resp.) The details of the hash functions are described in the extended paper since they not important for understanding the attack.
- *protocol* – the IP "next level" protocol number (for UDP, this value is 17). Nominally 8-bit field, extended to 32-bit by zero-filling the most significant bits.
- $\textsc{Random}(x), x > 0$ – a PRNG (a 96/128 bit Tausworthe Generator) which receives x as a parameter and provides a random integer in the range $[0, x)$. (We define $\textsc{Random}(0) = 0$). Note that $\textsc{Random}(1) = 0$.

The IP ID generation algorithm is defined in Algorithm 2. The procedure picks an index to a counter as a function of the source and destination IP address, the protocol and the key. It picks a random value which is smaller than or equal to the time that passed (measured in ticks, with tick frequency of f per second) since the last usage of this counter, increments the counter by this value, and outputs the result.

A_3**:** Starting with Linux 4.1, the net namespace of the kernel context, *net* (a 64-bit *pointer* in kernel space) is included in the hash calculation, conditional on a compilation flag `CONFIG_NET_NS` (which is on by default for Linux 4.1

[11] See function `__ip_select_ident` in https://elixir.bootlin.com/linux/v3.16/source/net/ipv4/route.c.

Algorithm 2 Linux IP ID Generation (A_1/A_2)

```
1: procedure GENERATE-IPID
2:     $i \leftarrow h(IP_{DST}, IP_{SRC}, protocol, key) \mod M$
3:     $hop \leftarrow 1 + \text{RANDOM}(t_{now} - \tau[i])$
4:     $\beta[i] \leftarrow (\beta[i] + hop) \mod 2^{16}$
5:     $\tau[i] \leftarrow t_{now}$
6:     return $\beta[i]$
```

and later, and for Android kernel 4.4 and later). The modification is for step 2, which now reads:

$$i \leftarrow h(IP_{DST}, IP_{SRC}, protocol \oplus g(net), key) \mod M$$

where $g(x)$ is a right-shift (by ρ bits) and a truncation function that returns 32 bits from x. We designate this algorithm as A_3.

To summarize, there are four flavors of IP ID generation (for short stateless protocol datagrams) in Linux:

1. A_0 - IP ID is always 0 (in ancient kernel versions)
2. A_1 / A_2 - Both versions use Algorithm 2, with the different implementations of h.
3. A_3 - Algorithm 2, adding net namespace to the calculation.

Of interest to us are algorithms A_1 to A_3. We focus mostly on UDP, as this is a stateless protocol which can be emitted by browsers.

The resolution f of the timer t in the algorithm is determined by the kernel compile-time constant `CONFIG_HZ`. A common value for older Android Linux kernels is 100(Hz). Newer Android Linux kernels (4.4 and above) use 300 or 100 (or rarely, 250). The default for Linux is $f = 250$.[12] In general, for tracking purposes, a lower value of f is better.

Note that *key* and *net* are generated during the operating system initialization, which, unlike Windows, happens during restart *and* during (shutdown+)start.

6.3 Setting the Stage

Our technique for tracking Android (and Linux) devices uses HTML5's WebRTC[1] both to discover the internal IP address of the device and to send multiple UDP packets. It works best when the WebRTC STUN [21] traffic is bursty. In order to analyze the effectiveness of the technique we investigated the following features, focusing on Android devices.

Android Versions and Linux Kernel Versions The Android operating system is based on the Linux kernel. However, Android versions do not map 1:1 to Linux kernel versions. The same Android version may be built with different Linux kernel versions by different vendors, and sometimes by the same vendor. Moreover, when an Android device updates its Android operating system, typically its Linux kernel remains on the same branch (e.g. 3.18.x). Android vendors also typically use somewhat old Linux kernels. Therefore, many Android devices in the wild still have Linux 3.x kernels, i.e. use algorithm A_1 or A_2.

Sending Short UDP Datagrams to Arbitrary Destinations, or "Set Your Browsers to STUN" The technique requires sending UDP datagrams from the browser to multiple destinations. The content of the datagrams is immaterial, as the tracker is interested only in the IP ID field. We use WebRTC (specifically – STUN) to send short UDP datagrams (with no control over their content) to arbitrary hosts. The `RTCPeerConnection` interface can be used to instruct the browser's WebRTC engine to use a list of presumably STUN servers, and even allows setting the UDP destination port per each host. The browser then sends STUN "Binding Request" (UDP short datagram) to the destination host and port.

To send STUN requests to multiple servers (in Javascript), create an array `A` of strings in the form `stun:`*host*`:`*port*, then invoke the constructor `RTCPeerConnection({iceServers: A}, ...)` in a regular WebRTC flow e.g. [13] (applying the fix from [8]).

Another option (specific to Google Chrome) is to send requests over gQUIC (Google QUIC) protocol, which uses UDP as its transport. This is less ideal since the traffic is less bursty, its transmission order isn't deterministic, and there is an overhead in HTTPS requests and in gQUIC packets.

Browser Distribution in Android We want to estimate the browser market share of "supportive" browsers (Chrome-like and Firefox) in the Android OS. Based on April 2018 figures for operating systems,[13] combined with mobile browsers distribution in April 2018,[14] we conclude that the Chrome-like browsers (Google Chrome, Opera Mini, Baidu, Opera) comprise 90% of the browser usage in Android. Adding Firefox (even though its STUN traffic is less bursty, Firefox can still be tracked at least for $f = 100$) gets this figure up to 92%.

Chrome's STUN Traffic Shape Chrome sends the STUN requests to the list of supposedly STUN servers, in bursts. A single burst may contain the full list of the requested STUN servers (in ascending order of destination IP address), or a subset of the ordered list (typically with a missing range of destination hosts). We measured 1014 bursts (to $L = 400$ destination IP addresses) emitted by a Google Pixel 2 mobile phone (Android 8.1.0, kernel 4.4.88), running Google

[12] https://elixir.bootlin.com/linux/v4.19/source/kernel/Kconfig.hz

[13] https://netmarketshare.com/operating-system-market-share.aspx

[14] https://netmarketshare.com/browser-market-share.aspx

Chrome 67 browser. The vast majority of bursts last between 0.1 seconds to 0.2 seconds, and the maximal burst duration was 0.548 seconds. Thus we use an upper bound of $\delta_L = 0.6$ seconds for a single burst duration.

Chrome emits up to 9 bursts with increasing time delays, at the following times (in seconds, where $t = 0$ is the first burst): 0, 0.25, 0.75, 1.75, 3.75, 7.75, 15.75, 23.75, 31.75.[15] We label these bursts $B_0, \ldots, B_8$ respectively, and we will be interested in B_4 and B_5, as they're sufficiently far from their neighbors. Thus, we are only interested in the first 8-9 seconds of the STUN traffic.

UDP Latency Distribution While WebRTC traffic is emitted by the browser in well defined, ordered bursts, one cannot assume the traffic will retain this "shape" when arriving to the destination servers. Indeed, even order among packets within a burst is not guaranteed at the destination. Understanding the latency distribution in UDP short datagrams is therefore needed in order to simulate the in-the-wild behavior, and consequently the efficacy of various tracking techniques. The latency of UDP datagrams is gamma-distributed according to [18] and [19]. However, for simplicity, we use normal distribution to approximate the in-the-wild latency distribution. On May 1st-6th 2018, we measured the latency of connections to a server in Microsoft Azure "East-US" location (in Virgina, USA) from 8 different networks located in Israel, almost 10,000km away. The maximum standard deviation was 0.081 seconds. Hereinafter, we will use a standard deviation value $\sigma = 0.1$ seconds as a worst case scenario for UDP jitter.

Packet Loss We identified two different packet loss scenarios:

- Packet loss during generation: the WebRTC packet stream (in Chrome-like browsers) is bursty in nature. In some bursts, we noticed large chunks of missing packets. These are quite rare (in the STUN traffic measurement experiment we got 29 such cases out of 1014 – 2.9%, though they are more common in Androids whose kernel is 4.x and have $f = 100$) and easily identified. We can safely ignore them because the tracker can detect a burst with a lot of missing packets, reject the sample and run the sampling logic again, or use a more sophisticated logic incorporating information from more than two bursts. Additionally, with $f = 100$ there are far less false pairs, which helps the analysis.
- Network packet loss: the UDP protocol does not guarantee delivery, and indeed packets get lost over the Internet. The loss rate is not high, however, and we estimate it to be $\leq 1\%$. This is also backed by research.[16]

[15] See https://chromium.googlesource.com/external/webrtc/+/master/p2p/base/stunrequest.cc).

[16] See http://www.verizonenterprise.com/about/network/latency/, and [4].

6.4 The Tracking Technique

The technique that we use is different than prior art techniques in focusing on bucket *collisions*. That is, in cases wherein UDP datagrams for two different destination IP addresses end up with IPID generated using the same counter.

The tracker needs to control L Internet IPv4 addresses, such that the IP-level traffic to these addresses (and particularly, the IP ID field) is available to the tracker. Ideally the IPs are all in the same network, so that they are all subject to the same jitter distribution. The tracker should be able to monitor the traffic to these IP addresses with time synchronization resolution of about 10 milliseconds (or less) - e.g. by having all the IPs bound to a single host.

With L different destination IP addresses and M buckets ($M = 2048$ in Algorithm 2), there are $\binom{L}{2}/M$ expected collisions, assuming no packet loss. In reality, the tracker can only obtain an *approximation* of this set. The goal is to reduce those false negatives and false positives to levels which allow assigning meaningful tracking IDs.

The basic property that enables the attacker to construct the approximate list is that in an IP ID generation the counter is updated by a random number which is smaller than 1 plus the multiplication of the timer frequency f and the time that passed since the last usage of that counter. Therefore for a true pair (IP^i, IP^j) where the IP ID generation for IP^i and IP^j used the same bucket (counter), the following inequality almost always holds:

$$0 < (IPID^j - IPID^i) \mod 2^{16} < f\Delta t + 10$$

(We use $f\Delta t + 10$ instead of $f\Delta t + 1$ to support up to 10 IPs colliding into the same bucket, as each collision may increment the counter by $\leq 1 + f\Delta t$ where Δt is from the *previous* collision. So the counter can end up incrementing no more than $f\Delta t + 10$ where Δt is the sum of the time difference between collisions, i.e. the time duration between the first collision and the last collision in the burst.)

Since we are looking at datagrams from the same burst we have an upper bound δ_L such that $\Delta t < \delta_L$, and therefore:

$$0 < (IPID^j - IPID^i) \mod 2^{16} < f\delta_L + 10$$

For two IP addresses which are *not* mapped to the same counter, the likelihood of this inequality to hold is only $\frac{(f\delta_L+10)-1}{2^{16}}$ which is $\ll 1$ when $f\delta_L \ll 2^{16}$. The key extraction algorithm (Section 6.6) will examine IP ID values in two different communication bursts, and this will further reduce the likelihood of a false positive. Note that the probability of a false positive pair in a given burst to survive into the next burst is roughly $\frac{f\delta_L+10}{f\Delta t} \approx \frac{\delta_L}{\Delta t}$ where Δt is the time between the consecutive bursts, whereas a true pair will occur in all bursts. Thus for the intersection of 2 consecutive bursts $\Delta t = 4$ seconds apart, the amount of false positives (in both bursts) will be ≈ 0.15 of their amount in a single burst.

6.5 Attack Phase 1 – Collecting Collisions

The tracking snippet needs to be rendered for at least 8.5 seconds, enough time for the browser to send the first 6 STUN bursts ($B_0,\dots,B_5$) – see Section 6.3. The tracking server splits the STUN traffic to bursts, based on the datagrams' time of arrival, and on the expected burst time offsets (see Section 6.3). For simplicity and ease of analysis, we henceforth only use traffic from bursts B_4 and B_5, which can be easily and unambiguously determined (since they are well separated in time from other bursts). We note that in some cases, requests in B_4 or in B_5 may be unsent, and in such cases we may need to resort to using e.g. B_3 and B_5 or similar combinations, but as long as these are "late" bursts (i.e. separated from their neighboring bursts by a enough σ units, where σ is the UDP jitter, see above), they can be separated without errors (or almost without errors) and the following analysis remains valid. If there are too many missing requests in a burst, the Tracking Server communicates with the Tracking Snippet, instructing it to retest the device.

Assuming no (or few) missing requests in B_4 and B_5, the Tracking Server starts analyzing the data per burst (in B_4 and B_5). For each burst the Tracking Server calculates a set of pair candidates by collecting pairs of IP addresses (IP^i, IP^j) for which $IP^i < IP^j$ and $0 < (IPID^j - IPID^i) \mod 2^{16} < \lambda_L$ where $\lambda_L = f\delta_L + 10$. It then identifies pairs which appear in the candidate sets of *both* bursts, and adds them to a set U of full candidates. This set forms a single *measurement* of a device. The tracker calculates the tracking ID based on U in Phase 2.

6.6 Attack Phase 2 – Exhaustive Key Search

In the second phase the tracking server runs an exhaustive search on the key space W where the key is 32 bits long for algorithms A_1 and A_2, 41 bits long for algorithm A_3 (Linux) and 48 bits for A_3 (Android). For each candidate key, the algorithm counts how many IP pairs in U are predicted by the candidate key. It is expected that only in one (the correct) key, this number will exceed a threshold ν, and in such case, this will be returned as the correct key (and the device ID). See Algorithm 3 for details (the algorithm uses the notation $h'(\dots,k) = h(\dots, protocol \oplus g(net), key)$ where k is split into $g(net), key$).

We assume here knowledge of the version of the algorithm (A) used – A_1, A_2 or A_3. For A_1 and A_2, the key space size is $|W| = 2^{32}$, and for A_3, it is 2^{41} for the x64 architecture and 2^{48} for the ARM64 architecture (see Section 6.7.)

As explained in Section 6.11, false positives ($|X| > 1$) are very rare – they can be handled but as this complicates the analysis logic, it is left out of the paper.

Attack run time Where $|U| = P$ pairs, the run time of Algorithm 3 is proportional to $|W|P$. P's distribution depends on f; Table 1 summarizes the expectancy and standard deviation for common f values. These were approximated by a computer simulation (100 million iterations.)

Algorithm 3 Exhaustive key search

```
1: procedure GENERATE-ID($U, IP_{src}$)    ▷ $U$ is defined in Section 6.5
2:     if $|U| < \nu$ then
3:         return ERROR
4:     $X \leftarrow \emptyset$
5:     for all $0 \le k < W$ do
6:         $Y \leftarrow \{(IP^i, IP^j) \in U \mid h'(IP^i, IP_{SRC}, k) = h'(IP^j, IP_{SRC}, k)\}$
7:         if $|Y| \ge \nu$ then
8:             $X \leftarrow X \cup \{k\}$
9:     if $|X| > 0$ then
10:        return $X$    ▷ Needs special treatment if $|X| > 1$
11:    else
12:        return ERROR
```

Table 1: Approximated P distribution

f [Hz]	$E(P)$	$\sigma(P)$	$\sigma(P)/E(P)$
100	50.59	7.39	0.146
250	65.47	8.60	0.131
300	70.45	8.79	0.125

Time/memory optimization When the number of devices to measure is much smaller than $|W|$ it is possible to optimize the technique for repeat visits. The optimization simply amounts to keeping a list Λ of already encountered *key* values (or $(g(net), key)$ values), and trying them first. If a match is found (i.e., this is a repeat visit), there is clearly no need to continue searching the rest of the key space. Otherwise, the algorithm needs to go through the remaining key space.

Targeted tracking Even if the key space W is too large to make it economically efficient to run large scale device tracking, it is still possible to use it for targeted tracking. The use case is the following: The tracking snippet is invoked for a specific target (device), e.g. when a suspect browses to a honeypot website. At this point, the tracker (e.g. law enforcement body) extracts the key, possibly using a very expensive array of processors, and not necessarily in real time. Once the tracker has the target's key, it is easy to test any invocation of the tracking snippet against this particular key and determine whether the connecting device is the targeted device. Moreover, if the attacker targets a single device (or very few devices), it is possible to reduce the number of IP addresses used for re-identifying the device, by using only IP addresses which are part of pairs that collide (into the same counter bucket) under the known device key. Thus we can use a single burst with as few as 5 IP pairs per device to re-identify the device. The dwell time in this case drops to near-zero.

6.7 The Effective Key Space in Attacking Algorithm A_3

In Algorithm A_3, 32 bits of the net namespace are extracted by a function we denote as $g()$, and are added to the calculation of the hash value. The attack depends on the effective keyspace size $|W| = |\{key\}| \times |\{g(net)\}| = 2^{32} \cdot |\{g(net)\}|$.

We analyzed the source code of Linux kernel versions 4.8 and above on x64, and 4.6 and above on ARM64, and found that if KASLR is turned off then the effective key space size is 32 bits in both x64 and ARM64. If KASLR is turned on, then the effective key space size is 41 bits in x64 and 48 bits in ARM64.

6.8 KASLR Bypass for Algorithm A_3

By obtaining $g(net)$ as part of Attack Phase 2 (Section 6.6), the attacker gains 32 bits of the address of the `net` structure. In single-container systems such as desktops and mobile devices, this `net` structure resides in the `.data` segment of the kernel image, and thus has a fixed offset from the kernel image load address. In default x64 and ARM64 configurations, the 32 bits of $g(net)$ completely reveal the random KASLR displacement of `net`. This suffices to reconstruct the kernel image load address and thus fully bypass KASLR.

6.9 Optimal Selection of L

Since IP addresses are at premium, we choose a minimal integer number L of IP addresses such that at the point ν where $Prob(FN) + Prob(FP)$ is minimal, $Prob(FP) + Prob(FN) \leq 10^{-6}$. We assume $f = 300$ (worst case scenario). For simplicity, at this stage we neglect packet loss, and assume that $\delta_L = \frac{L}{400}\delta_{400}$ (we assume $\delta_L \propto L$, and we measured δ_{400}). For false negatives, we use the Poisson approximation of birthday collisions [3] with $\lambda = \binom{L}{2}/M$. Therefore:

$$Prob(FN) \approx \sum_{i=0}^{\nu-1} \frac{\lambda^i e^{-\lambda}}{i!}$$

For false positives, we also assume that a burst contains the average number of false pairs and true pairs $A = \lfloor \frac{f\delta_L+10}{f\Delta t}\binom{L}{2}\frac{f\delta_L+10}{2^{16}} + \frac{\binom{L}{2}}{M} \rfloor$. We note that the probability for a single false key to match exactly k pairs is $\binom{A}{k}(\frac{1}{M})^k(1-\frac{1}{M})^{A-k}$. The probability of $|W|-1$ false keys to generate at least one false positive key is therefore:

$$Prob(FP) \approx 1 - \Big(\sum_{i=0}^{\nu-1}\binom{A}{i}(\frac{1}{M})^i(1-\frac{1}{M})^{A-i}\Big)^{|W|-1}$$

Assuming $|W| = 2^{48}$ (worst case – Android), we enumerated over all ν values for each L in $\{200, 250, \ldots, 500\}$ to find the optimal ν (per L). We found that $L = 400$ (with $\nu = 11$) is the minimal "round" L satisfying $Prob(FP) + Prob(FN) \leq 10^{-6}$ at its optimal ν.

6.10 A More Accurate Treatment for $L = 400$

Using a computer simulation, we approximated the distributions of all collisions $p_A(n)$ (using 10^8 simulation runs), and of true collisions $p_T(n)$ (using 10^9 simulation runs). The simulations took into account 1% packet loss. With these, we can calculate more accurate approximations:

$$Prob(FN) \approx \sum_{i=0}^{\nu-1} p_T(i)$$

$$Prob(FP) \approx 1 - \sum_n p_A(n)\Big(\sum_{i=0}^{\nu-1}\binom{n}{i}(\frac{1}{M})^i(1-\frac{1}{M})^{n-i}\Big)^{|W|-1}$$

(We use the convention $\binom{n}{k} = 0$ where $k > n$.) We enumerated over values $1 \leq \nu \leq 20$ for $L = 400$ and $|W| = 2^{48}$ (worst case – Android.) The minimal $Prob(FP) + Prob(FN)$ is at $\nu = 11$, where $Prob(FP) = 6.2 \times 10^{-10}$ and $Prob(FN) = 4.2 \times 10^{-8}$. We get the same optimal ν value for $L = 400$ as we got in Section 6.9, which means that the approximation steps we took there are reasonable.

6.11 Practical Considerations

Controlling packets from the browser As explained in Section 6.3, it is possible to emit UDP traffic to arbitrary hosts and ports using WebRTC. The packet payload is not controlled. The tracker can use the UDP destination port in order to associate STUN traffic to the same measurement.

Synchronization and packet transmission/arrival order Unlike the Windows technique, in the Linux/Android tracking technique there is no need to know the exact transmission order of the packets within a single burst.

False positives and false negatives Using a computer simulation with $L = 400$ destination IP addresses, a burst length of $\delta_L = 0.6$ seconds, and packet loss rate of 0.01, we calculated an approximation of for the false negative rate of 4.2×10^{-8} for $\nu = 11$, and an approximation for the false positive rate of 6.2×10^{-10}. These approximations were computed assuming $|W| = 2^{48}$ (worst case – Android). See Section 6.10 for more details.

Device ID collisions The expected number of pairs of devices with colliding IDs, due to the birthday paradox, and given R devices and a key space of size $|W|$, is $\binom{R}{2}/|W|$. For Algorithms A_1 and A_2 the key space size is $|W| = 2^{32}$, and will cause device ID collisions once there are several tens of thousands of devices. For $R = 10^6$ this will affect 0.00023 of the population (2 out of every 10,000 devices). For Alg. A_3, the key space size (with KASLR) is $\geq 2^{41}$, so collisions start showing up with R in the millions. Even for $R = 128 \cdot 10^6$, collisions affect only 0.00006 of the population.

Dwell time In order to record B_5, the snippet page needs to be loaded in the browser for 8-9 seconds. Navigating away from the page will immediately terminate the STUN traffic.

Environment factors All the UDP-related topics in Section 4.8 are applicable as environment factors on the Linux/Android tracking technique.

Longevity The device ID remains valid as long as the device is not shutdown or restarted. Mobile devices are rarely shut down, and are typically restarted only on updates, which happen once every several months, or even less frequently.

Scalability The attack is scalable. Device ID collisions are rare even with many millions of devices (see above). False positives and false negatives are also rare (less than 4.3×10^{-8} combined). From a resource perspective, the attack uses a fixed number of IPs and servers, and a fixed-size RAM/disk. The required CPU power is proportional to the number of devices measured per time unit. Network consumption per test is negligible – approx. 13.5KB/s (at the IP level) during measurement.

6.12 Possible Countermeasures

Increasing M Changing the algorithm to use a larger number M of counters, will reduce the likelihood of pairs of IP addresses using the same counter. In response to such a change the tracker can increase the number L of IP addresses that is uses. The expected number of collisions is $\binom{L}{2}/M$, and therefore increasing M by a factor of c requires the attacker to increase L by only a factor of $\sqrt{c}$.

On the other hand, δ_L also grows (probably linearly in L), and when $f\delta_L \geq 2^{16}$ no information is practically revealed to the tracker. It is probably safe to assume that the tracker can handle an increase of L by a factor of $\times 10$, which means that in order to stop the attacker the IP ID generation algorithm must increase M by more than $\times 100$, making it too memory expensive to be practical.

Increasing the key size (W) This can be an effective counter-measure for the exhaustive search phase, though the pair collection phase is unaffected by it. Yet some choices of the hash function h might still allow fast cryptanalysis.

Strengthening h Our analysis does not rely on any property of the hash function h, except that it is more-or-less uniform. Thus, changing h will not affect our results.

Replacing the algorithm See the last item in Section 4.9.

7 Experiment – Attacking Linux and Android Devices in the Lab

In order to verify that we can extract the key used by Linux and Android devices, we need to control hundreds of IP addresses. Controlling such a magnitude of Internet-routable IP addresses was logistically out of scope for this research. Therefore we had to settle for an in-the-lab setup, which naturally limited the number of devices we could test.

7.1 Setup

We connected the tested devices to our own WiFi access point, which advertised our laptop as a network gateway. Then we launched a Chrome-like browser inside the Linux/Android device, and navigated to a page containing a tracking snippet. The tracking snippet used WebRTC to force UDP traffic to a list of $L = 400$ hosts, and this traffic passed through our laptop (as a gateway) and was recorded.

We then ran the collision collection logic (Phase 1), and fed its output (IP pairs whose IP IDs collide) to the exhaustive key search logic (Phase 2). For KASLR-enabled devices, we also provided the algorithm with the offset (relative to the kernel image) of `init_net`, which we extracted from the kernel image file given the build ID (can be inferred e.g. from the `User-Agent` HTTP request header). We expected that the algorithm will output a single key, which will match a large part of the collisions.

7.2 Results

We tested 2 Linux laptops and 6 Android devices, together covering the vast majority of operating system and hardware parameters that regulate the IP ID generation. The results from all tests were positive - our technique extracted a single key and a kernel address of `init_net` where applicable (which was identical to the address in `/proc/kallsyms`). Note that due to hardware availability constraints, for the Pixel 2XL case ($|W| = 2^{48}$), we provided the algorithm with the correct 16 bit kernel displacement to reduce the key search to 2^{32}. Table 2 provides information about the common kernel versions, their parameter combinations and the tested devices.

The Attack time column is the extrapolated attack time in seconds with 10,000 Azure B1s machines, based on $E(P_f)$ from Table 1, i.e. the average attack time is $r \cdot |W| \cdot E(P_f)$ where r is the time it takes a single B1s machine to test a single key with a single pair, divided by 10,000. The standard deviation of the attack time for a given f is $r \cdot |W| \cdot \sigma(P_f)$, which is $\sigma(P_f)/E(P_f)$ in Table 1 times the average attack run time in Table 2. From a calibration run (single B1s machine, 10 pairs, 2^{32} keys, 294.83 seconds run time) we calculated $r = 6.8645 \times 10^{-13}$, and populated the Attack Time column in Table 1 with $r \cdot |W| \cdot E(P_f)$.

Applicability in-the-wild While our tests were carried out in the lab, we argue that the results are representative of an in-the-wild experiment with the same devices. We list the following potential differences between in-the-lab and in-the-wild experiment, and for each difference, we note why our experiment can be projected to an in-the-wild scenario.

- Packet loss: our technique is not sensitive to packet loss. We ran false positive/negative computer simulations (assuming 1% packet loss) supporting this fact.

Table 2: Common Linux/Android Kernels and Their Parameter Combinations

O/S	Kernel Version	Alg.	f [Hz]	KASLR	NET_NS	ρ	$\log_2 \lvert W \rvert$	Tested System	Attack Time [s]
Linux (x64)	4.19+	A_3	250	Yes	Yes	12	41	Dell Latitude E7450 laptop	99
Linux (x64)	4.8-4.18.x	A_3	250	Yes	Yes	6	41	Dell Latitude E7450 laptop	99
Android (ARM64)	4.4.56+, 4.9, 4.14	A_3	300/ 100	Yes	Yes	6/7	48	Pixel 2XL ($\rho = 6$)	13,612/ 9,775
Android (ARM64)	3.18.17+ 3.4.109+	A_2	100	No	No	Don't care	32	Redmi Note 4 Xiaomi Mi4	0.15
Android (ARM64)	3.18.0-3.18.6 3.10.53+ 3.4.103-3.4.108	A_1	100	No	No	Don't care	32	Samsung J7 prime Samsung S7 Meizu M2 Note	0.15

- Network latency: our technique is not sensitive to network latency (which is just a constant time-shift, from our perspective).
- UDP jitter: this only affects correctly splitting the traffic into bursts. Our technique uses the "late" bursts, thus assuring that the bursts are well separated time-wise and that a jitter of $\sigma = 0.1$s does not affect tracking.
- Network interference (IPID modification): this issue was already evaluated in-the-wild in the Windows experiment, and the Windows results can be applied to the Linux/Android use case.
- Packet reordering (within a burst): Our technique does not rely on packet order within a burst.

Thus we conclude that our results (and henceforth, the practicality of our technique) are applicable in-the-wild.

8 Conclusions

Our work demonstrates that using non-cryptographic random number generation of attacker-observable values (even if the values themselves are not security sensitive), may be a security vulnerability in itself, due to an attacker's ability to extract the key/seed used by the algorithm, and use it as a fingerprint of the system.

We stress that any replacement cryptographic algorithm must not be hampered by using a key that is too short, in order to avoid a key enumeration attack. Also, as a security measure, we strongly recommend generating unique keys for such cryptographic usage, without resorting to using secret data that is used for other purposes (which – in case of a cryptographic weakness in the algorithm – can leak out).

9 Acknowledgements

This work was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

We would like to thank the anonymous reviewers for their feedback, Assi Barak for his help to the project, as well as Avi Rosen, Sharon Oz, Oshri Asher and the Kaymera Team for their help with obtaining a rooted Android device.

References

[1] B. Aboba, D. Burnett, T. Brandstetter, C. Jennings, A. Narayanan, J.-I. Bruaroey, and A. Bergkvist. WebRTC 1.0: Real-time communication between browsers. Candidate recommendation, W3C, June 2018. https://www.w3.org/TR/2018/CR-webrtc-20180621/.

[2] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *ACM CCS '13*, pages 1129–1140, 2013.

[3] R. Arratia, L. Goldstein, and L. Gordon. Two moments suffice for poisson approximations: The chenstein method. *Ann. Probab.*, 17(1):9–25, 01 1989.

[4] D. Baltrunas, A. Elmokashfi, A. Kvalbein, and Ö. Alay. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference and Workshops*, pages 225–233, 2016.

[5] S. M. Bellovin. A technique for counting Natted hosts. In *2nd SIGCOMM Workshop on Internet Measurement*, pages 267–272, 2002.

[6] Y. Gilad and A. Herzberg. Fragmentation considered vulnerable. *ACM Trans. Inf. Syst. Secur.*, 15(4):16:1–16:31, Apr. 2013.

[7] F. Gont. Security Implications of Predictable Fragment Identification Values. RFC 7739, Feb. 2016.

[8] Google. Issue 928273: unified plan breaks rtp datachannels with empty datachannel label, Feb. 2019.

[9] A. Herzberg and H. Shulman. Fragmentation considered poisonous. *CoRR*, abs/1205.4011, 2012.

[10] T. Hudek and D. MacMichael. RSS hashing functions.

[11] Information Sciences Institute (University of Southern California). Internet Protocol. RFC 791, Sept. 1981.

[12] A. Janc and M. Zalewski. Technical analysis of client identification mechanisms. https://www.chromium.org/Home/chromium-security/client-identification-mechanisms.

[13] M. Khan. RTCDataChannel for beginners. https://www.webrtc-experiment.com/docs/rtc-datachannel-for-beginners.html, 2013.

[14] A. Klein. Predictable javascript math.random and http multipart boundary string. http://www.securitygalore.com/site3/math_random_and_multipart_boundary.

[15] A. Klein. Detecting operation of a virtual machine (US patent 9384034), July 2016.

[16] J. Knockel and J. R. Crandall. Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, 2014.

[17] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *5th ACM SIGCOMM Conf. on Internet Measurement*, IMC '05, pages 33–33, 2005.

[18] H. Li and L. Mason. Estimation and simulation of network delay traces for voip in service overlay network. *2007 International Symposium on Signals, Systems and Electronics*, pages 423–425, 2007.

[19] S. Maheshwari, K. Vasu, S. Mahapatra, and C. S. Kumar. Measurement and analysis of UDP traffic over wi-fi and GPRS. *CoRR*, abs/1707.08539, 2017.

[20] G. Maone. NoScript. https://noscript.net/.

[21] P. Matthews, J. Rosenberg, D. Wing, and R. Mahy. Session Traversal Utilities for NAT (STUN). RFC 5389, Oct. 2008.

[22] A. Melnikov and I. Fette. The WebSocket Protocol. RFC 6455, Dec. 2011.

[23] R. Menscher. Exploiting Windows' IP ID randomization bug to leak kernel data and more (CVE-2018-8493). https://menschers.com/2018/10/30/what-is-cve-2018-8493/, November 2018.

[24] Microsoft. CVE-2018-8493 | windows TCP/IP information disclosure vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8493.

[25] L. Orevi, A. Herzberg, and H. Zlatokrilov. DNS-DNS: DNS-based de-nat scheme. In *Cryptology and Network Security CANS*, pages 69–88, 2018.

[26] D. Reed, P. S. Traina, and P. Ziemba. Security Considerations for IP Fragment Filtering. RFC 1858, Oct. 1995.

[27] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[28] F. Salutari, D. Cicalese, and D. Rossi. A closer look at ip-id behavior in the wild. In *International Conference on Passive and Active Network Measurement (PAM)*, Berlin, Germany, Mar. 2018.

[29] H. Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *13th Workshop on Privacy in the Electronic Society*, WPES '14, pages 191–200, 2014.

[30] M. J. Silbersack. darwin-xnu/bsd/netinet/ip_id.c. https://opensource.apple.com/source/xnu/xnu-4570.41.2/bsd/netinet/ip_id.c.

[31] H. Wramner. Tracking users on the world wide web. http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/wramner_henrik_11041.pdf, 2011.

[32] M. Zalewski. *Silence on the Wire*. No Starch Press, 2005.

A Details of the Attack on Windows

A.1 Practical Considerations

A.1.1 Controlling Packets from the Browser

UDP: As explained in Section 6.3, it is possible to emit UDP traffic to arbitrary hosts and ports using WebRTC. The packet payload is not controlled. The tracker can use the UDP destination port in order to associate STUN traffic to the same measurement.

TCP: WebSocket [22] emits TCP traffic in a controlled fashion once a circuit is established, thus can be used by the snippet to fully control packet transmission. The downside of using TCP-based protocols is the TCP-level retransmission, which can introduce loss of synchronization between the device and the server side, regarding how many packets were sent. The tracker can use the packet payload to mark packets that belong to the same measurement.

Table 3: Common tail length probability - measured with 1000 randomly chosen sets of 30 IPs ($J = 6, G = 12, Q = 3$), 10,000 tests each (10^7 tests altogether).

Common tail [bits]	Prob.	Common tail [bits]	Prob.
45	0.9937579	42	2.6×10^{-5}
44	0.0058328	41	2.9×10^{-6}
43	0.0003783	≤ 40	2.1×10^{-6}

A.1.2 Packet Transmission Order

We encountered cases in the wild where the packet payload generation order is not identical to packet transmission order. Specifically, Microsoft IE and Edge are prone to this behavior. This is only relevant in the same class B network (since there the original extraction algorithm makes an assumption on the order of the packets). Therefore, the tracker should try all possible permutations of packet order (per class B network IPs). In Phase 1, this means enumerating over all $\pi \in S_J$ ($J!$ permutations). For each permutation, use the following definition of D^j (instead of the original one):

$$D^j = (\text{Vec}(IPID^j - (\beta_0 + \pi(j)) \mod 2^{15}) \oplus$$
$$\text{Vec}(IPID^0 - (\beta_0 + \pi(0)) \mod 2^{15}))_{17,\ldots,31}$$

It follows that enumerating over all possible orders will increase the run time of Phase 1 by a factor of $J!$. In Phase 2, for each pair of IP addresses, there are only 2! permutations, and since the elimination is so powerful, this will only affect the run time due to the first pair, i.e. will double the run time.

A.1.3 Handling False Positives

The issue of false positive keys is covered in the extended paper. As mentioned there, the vast majority of false positive keys only differ from the correct key by a few leftmost bits. Table 3 demonstrates that with an optimal choice of 30 IP addresses, if the tracker keeps only 41 bits of the key tail, he/she will get multiple keys with probability 2.1×10^{-6}, which is sufficiently small even for a large scale deployment.

In the case where multiple keys are emitted by the algorithm (even after truncation, e.g. to 41 bits), two strategies can be applied: either (a) determining that this particular device cannot be assigned an ID (at the price of losing 2.1×10^{-6} of the devices); or (b) assigning multiple IDs to the device (which makes tracking the device more complicated and more prone to ID collisions).

A.1.4 Handling False Negatives and Interference

It is important to note that while there may be false positives, there are no algorithmic false negatives, i.e. the algorithm always emits the correct key (possibly along with incorrect keys), given the correct data. However, it is possible for the algorithm to receive incorrect data, in the sense that IP IDs are provided which are not (even after re-ordering) derived from an incrementing counter – i.e. there are "gaps" in the counter values associated with the IP IDs. This can happen either due to packet loss or due to interference.

Packet loss: In TCP, when a packet from the client to the server is lost, the client will note a missing ACK and will eventually retransmit the original data (with incremented IP ID). This will cause a gap in the counter values, which can be enumerated by the analysis logic (our analysis logic does not currently implement this). Another packet loss scenario is wherein the ACK packet from the server to the client is lost, and the client retransmits the original data. The server however receives two such packets, and can simply discard the one with incremented IP ID. Thus the "problematic" scenario is the one wherein the original data packet is lost.

Interference: Theoretically, an unrelated packet sent by the Windows device in between measurement packets may interfere with the measurement. However, this is very unlikely. First, the interfering packet must fall into the same counter bucket as that of the measurement packets – this happens with probability of $1/8192$ for a given bucket. Second, the timing is delicate – the interfering packet should be sent between the first and the last measurement packet. This time window is below 1 second, so overall the likelihood for interference is very low. When such an interference happens, it creates a gap in the IP ID values, which can be addressed as explained below.

Addressing "gaps": The analysis logic can compensate for up to l lost packets in the first class B network ($g = 0$) by enumerating over all possible $\sum_{d=0}^{l} \binom{J-2+d}{J-2}$ gap configurations (each addendum counts all weak compositions of d into $J-1$ parts). In our experiment, we measured $l = 4$ for some "difficult" networks, so for $J = 6$ there are 126 gap configurations, thus a $\times 126$ factor in the runtime. Note that a gap in the transmissions for $g > 0$ is much easier to handle, as $(IPID^{g,1} - IPID^{g,0})$ in eq (5) may now take values in $\{1, \ldots, l+1\}$, so there is no runtime factor for this case. When the total gap space is larger than l, the algorithm will yield no key. In such a case, the server can instruct the snippet to run another test. Therefore the actual false negative probability can be reduced to as small as necessary.

A.2 Optimizing the IP Set for Minimum False Positives

Since (from Table 3) keys with flipped K_{18} are the source of most false positives, the tracker should choose a set of IP addresses that minimizes (over Q) this false positive probability, $2^{-(J-1)-Q} + 2^{-(G-Q)}$ (this calculation can be found in the extended paper.) The said minimum is at $Q = G-(J-1)/2$, and yields false positive leading term of $2 \cdot 2^{-\frac{G+J-1}{2}}$. For $G = 12, J = 6$, the optimum is at $Q = 3$ or $Q = 4$.