A Taxonomy and Comparative Analysis of IPv4 Identifier Selection Correctness, Security, and Performance

JOSHUA J. DAYMUDE, Arizona State University, USA
ANTONIO M. ESPINOZA, Eastern Washington University, USA
HOLLY BERGEN, Arizona State University, USA
BENJAMIN MIXON–BACA, Arizona State University, USA
JEFFREY KNOCKEL, Bowdoin College, USA
JEDIDIAH R. CRANDALL, Arizona State University, USA

The battle for a more secure Internet is waged on many fronts, including the most basic of networking protocols. Our focus is the *IPv4 Identifier* (IPID), an IPv4 header field as old as the Internet with an equally long history as an exploited side channel for scanning network properties, inferring off-path connections, and poisoning DNS caches. This article taxonomizes the 25-year history of IPID-based exploits and the corresponding changes to IPID selection methods. By mathematically analyzing these methods' correctness and security and empirically evaluating their performance, we reveal recommendations for best practice as well as shortcomings of current operating system implementations, emphasizing the value of systematic evaluations in network security.

CCS Concepts: • **Security and privacy → Network security**; • **Networks → Routing protocols**.

Additional Key Words and Phrases: IP identifiers, side channels, comparative analysis, probability theory

1 INTRODUCTION

There is ongoing interplay between operating system (OS) developers implementing basic networking protocols, such as IP and TCP, and security researchers discovering side channels and vulnerabilities in those diverse implementations. A prime example is IP fragmentation and reassembly, a staple of Internet functionality for over forty years [64]. When an IPv4 packet is too large for the next link of its routing path, it is *fragmented*; a destination machine then *reassembles* fragments it receives based on their *IPv4 Identifiers* (IPIDs), a 16-bit header field. OSes are free to assign IPIDs

Authors' addresses: Joshua J. Daymude, Arizona State University, Biodesign Center for Biocomputing, Security and Society, School of Computing and Augmented Intelligence, 727 E. Tyler St., Tempe, AZ, 85281, USA, jdaymude@asu.edu; Antonio M. Espinoza, Eastern Washington University, College of Science, Technology, Engineering, and Mathematics, 601 E Riverside Ave., Spokane, WA, 99202, USA, aespinoza17@ewu.edu; Holly Bergen, Arizona State University, Biodesign Center for Biocomputing, Security and Society, School of Computing and Augmented Intelligence, 727 E. Tyler St., Tempe, AZ, 85281, USA, sbergen2@asu.edu; Benjamin Mixon–Baca, Arizona State University, Biodesign Center for Biocomputing, Security and Society, School of Computing and Augmented Intelligence, 727 E. Tyler St., Tempe, AZ, 85281, USA, bmixonba@asu.edu; Jeffrey Knockel, Bowdoin College, Department of Computer Science, Brunswick, ME, 04011, USA, j.knockel@bowdoin.edu; Jedidiah R. Crandall, Arizona State University, Biodesign Center for Biocomputing, Security and Society, School of Computing and Augmented Intelligence, 727 E. Tyler St., Tempe, AZ, 85281, USA, jedimaestro@asu.edu.

159

to packets however they want, so long as they avoid causing ambiguous reassembly by sending multiple packets with the same IPID to the same destination close in time.

Despite early concerns about IP fragmentation's performance impacts [41] and later concerns about its vulnerabilities [30, 34], its central role in the Internet—and in DNS in particular—continues to drive the discovery of new vulnerabilities in IPID selection methods. Since 1998, numerous exploits have abused these methods to poison DNS caches [34, 61, 73], hijack TCP connections [25, 26, 29], launch denial of service attacks [53], scan ports [2, 71], detect and measure connections [1, 47], detect Internet censorship [22], and create covert channels [44]—all from off-path vantage points requiring nothing more than an active Internet connection. Understanding the aims and necessary conditions of these diverse exploits is critical for proactively securing the menagerie of existing IPID selection methods from further misuse.

The purpose of this survey is two-fold. First, we taxonomize the 25-year history of off-path IPID-based exploits and the subsequent changes to IPID selection methods across all major OSes. This taxonomy has two levels, categorizing exploits primarily by their mechanisms (how does an exploit use IPIDs?) and secondarily by their end goals (what does the exploit do?). Specifically, we demonstrate that despite the number and apparent diversity of IPID-based exploits, there are only two core mechanisms in use: (1) *probe comparisons* that track changes in IPIDs over time to infer information about other machines, and (2) *fragment injections* that use guessed IPIDs to trick a target machine into replacing legitimate fragments with malicious ones during reassembly. For each mechanism, we identify the archetypal approach and resulting necessary conditions for success, thus distilling the key properties of (in)secure IPID selection methods.

Second, we perform a unifying comparative analysis of all seven major IPID selection methods across three qualities: *correctness* (does the method supporting unambiguous reassembly by avoiding IPID collisions?), *security* (are sequences of IPID values sufficiently difficult to predict?), and *performance* (what are the method's time and space complexities?). By parameterizing this analysis in terms of a machine's expected rate of traffic and number of CPUs, we reveal tradeoffs among the selection methods for different use cases (e.g., a high-traffic DNS server vs. a low-traffic home computer). Among these evaluations, the most surprising is that globally incrementing selection—the first, simplest, and most dismissed of all IPID selection methods—is in fact the most collision-avoidant, secure, and performant choice for non-connection-bound packets when the rate of outgoing packets is high. We conclude by proposing a new approach to IPID selection that shifts away from single-method implementations that necessarily compromise at least one of correctness, security, or performance across different use cases, instead embracing multiple implementations that end users can choose from (e.g., as a network setting) according to their specific needs.

The remainder of this paper is organized as follows. Section 2 introduces IP fragmentation and reassembly, the seven primary IPID selection methods, and their current OS implementations. Section 3 reviews the history of exploits using IPIDs for measurements or malicious attacks and the corresponding changes to selection methods. Section 4 presents our comparative analysis of IPID selection methods' correctness, security, and performance, summarized in Table 1. We synthesize these evaluations as recommendations for a new approach to IPID selection in Section 5; readers seeking to quickly digest "what's new" should skip to this section and refer to Fig. 9 as a visual summary of our proposed approach. Finally, we conclude in Section 6.

2 BACKGROUND

We begin by reviewing how IPIDs are used, the established methods for selecting IPIDs, and the current OS implementations of those selection methods.
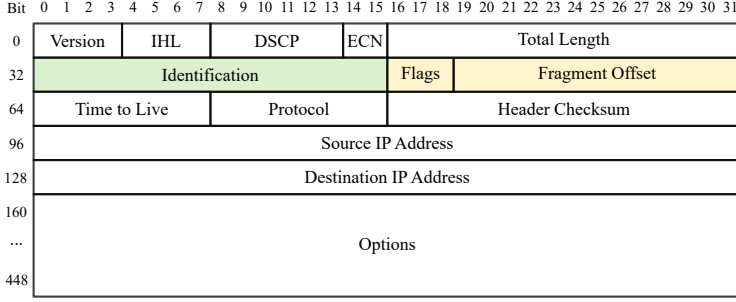
Bit	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

0	Version	IHL	DSCP	ECN	Total Length
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	Header Checksum	
96	Source IP Address				
128	Destination IP Address				
160 ... 448	Options				

Fig. 1. *Anatomy of an IPv4 Packet Header.* The 16-bit Identification field is the packet's IPID. The Flags and Fragment Offset fields are also used in IP fragmentation and reassembly.

2.1 IP Packet Fragmentation and Reassembly

The IPID is a 16-bit IPv4 header field used for packet reassembly (see Fig. 1). Every IPv4 packet is assigned an IPID by its sender. If ever a packet is too large for some link of its routing path, an intermediate router breaks it into *fragments* that inherit the IPID of their packet. As the packet's destination receives these fragments in its *reassembly buffer* (also known as the *fragmentation cache*), it uses their IPIDs to group them by packet and inspects their *Flags* and *Fragment Offset* IPv4 header fields to determine whether all of the packet's fragments have arrived and in what order they should be reassembled. This continues until the packet is reassembled or a timeout of 15–120 seconds is reached [10, 64]. The main restriction concerning IPID selection is that IPIDs must be "unique" to avoid ambiguous reassembly [64].

IP fragmentation is not always necessary. For example, if a sender knows a path's maximum transmission unit (MTU)—e.g., as part of a TCP connection—it can resize its packets prior to sending them. Such packets can be made *atomic* to discourage IP fragmentation, setting their *Don't Fragment* (DF) flag to TRUE, their *More Fragments* (MF) flag to FALSE, and their *Fragment Offset* to 0. IPIDs have no utility in atomic packets and are thus ignored and allowed to be any value [70].

In IPv6, senders are "strongly recommended" to avoid fragmentation by either implementing path MTU discovery and resizing packets accordingly or limiting their packets to 1,280 octets, the IPv6 minimum link MTU [19]. When fragmentation does occur in IPv6, it is performed only by senders; intermediate routers never fragment IPv6 packets. If a router determines that an IPv6 packet's size exceeds the next link's MTU, it drops the packet and sends an ICMPv6 "Packet Too Big" message back to the sender. To send an IPv6 packet whose size exceeds the path MTU, a sender can split the packet into fragments and send each fragment as a separate IPv6 packet including the optional *Fragment* extension header. This extension header includes a 32-bit identification value used by the receiver during reassembly, analogous to the IPID in IPv4. We primarily focus on IPv4's IPID in this survey, but will briefly return to IPv6 fragmentation in Section 3.3.

2.2 IPID Selection Methods

In this section, we detail the seven existing methods for selecting packets' IPIDs. These methods fall into two categories: counter-based (Section 2.2.1) and PRNG-based (Section 2.2.2).

2.2.1 Counter-Based Selection Methods.

- *Globally incrementing* IPID selection (Algorithm 1, Lines 1–4) is the earliest and simplest method. Each machine maintains a single, sequentially-incrementing, 16-bit counter for

all IPID generation. Whenever a machine sends a packet, it increments its global counter (modulo 2^{16}) and uses the resulting value as the packet's IPID.

- *Per-connection* IPID selection (Algorithm 1, Lines 5–8) generalizes globally incrementing selection by maintaining one counter per active *connection*, i.e., a 5-tuple specifying the source and destination IP addresses, source and destination ports, and protocol number. Each connection counter is initialized with a random 16-bit value when the connection is established and subsequently assigns IPIDs to packets by incrementing by one, modulo 2^{16}.

- *Per-destination* IPID selection (Algorithm 1, Lines 9–16) is analogous to per-connection selection, but with one counter per pair of source and destination IP addresses.

- *Per-bucket* IPID selection (Algorithm 1, Lines 17–31) is similar to per-connection and per-destination selection, but with one counter per "bucket". Packets are typically mapped to buckets by hashing their IP header information (e.g., source and destination IP addresses or the protocol number) with cryptographic information like secret keys. Some implementations use sequentially-incrementing bucket counters, analogous to the above methods, while others inject some stochasticity. These variations are detailed by OS in Section 2.3.

2.2.2 PRNG-Based Selection Methods. Methods based on *pseudorandom number generation* (PRNG) select IPIDs at random from among the 2^{16} possible values, optionally employing some mechanism for avoiding recently-selected IPIDs whose reuse could cause ambiguous reassembly. Naturally, these methods' quality depend on unbiased, secure PRNG algorithms drawing from good sources of entropy. Cryptanalysis of PRNGs (e.g., [20, 32]) is beyond the scope of this survey, but we will discuss examples of poor-quality PRNGs affecting IPID selection in Sections 3.1.2 and 3.2.1.

- One method for generating non-repeating random IPIDs is with a *searchable queue* storing the last $k \geq 0$ unique IPIDs (Algorithm 1, Lines 32–38). When an IPID is requested, the PRNG generates 16-bit values until one is found that is not already in the queue; this new value is then inserted into the queue and used as the next IPID. The size of the searchable queue varies by implementation, but is typically chosen between 4,096 and 32,768 (i.e., 2^{12} and 2^{15}) to provide a good tradeoff between entropy and non-repetition.

- Another method uses an *iterated Knuth shuffle* [48], also known as the Durstenfeld algorithm for random permutations or the Fisher–Yates shuffle (Algorithm 1, Lines 39–45). Initially, a random permutation of all 2^{16} IPID values is generated by a standard Knuth shuffle. An index i marks the current "head" of the (cyclic) permutation. When an IPID is requested, i is incremented and the i-th value in the permutation is used. That value is then swapped into a position chosen uniformly at random from $[i - (2^{16} - k - 1) \bmod 2^{16}, i]$ (i.e., among the previous $2^{16} - k$ positions in the cyclic permutation, including its own), ensuring that it will not be selected again within the next k requests for IPIDs.

- *Pure PRNG selection* simply generates IPIDs from among the 2^{16} possible values uniformly at random, possibly repeating IPIDs. This is functionally equivalent to the previous two methods with $k = 0$, but has no need for their data structures ensuring non-repetition.

2.3 Current Implementations

In this section, we survey current implementations of IPID selection across popular OSes.

Windows. Source code for Windows 8, 10, and 11 is not publicly available, though reverse-engineering efforts revealed that until related issues were patched in 2019, Windows 8 and 10 used per-bucket IPID selection based on the Toeplitz hash function with 8,192 (i.e., 2^{13}) buckets [46]. Recent analysis shows that Windows Server (v1903) uses per-destination IPID selection [44]. Specifically, Windows maintains a "PathSet" (i.e., a hash table) of sequentially incrementing counters

indexed by source and destination IP address pairs, adding a new counter whenever a packet's destination is not already in the PathSet. The PathSet's size is checked every 0.5 s: if it has grown beyond its "purge threshold"—4,096 (i.e., 2^{12}) counters in Windows 10 or 32,768 (i.e., 2^{15}) in Windows Server—or more than 5,000 counters have been added since the last check, a "purge sequence" removes up to max{1,000, # added since last check} "stale" counters. Counters are considered "stale" depending on when they were last accessed and the PathSet's size: if the PathSet's size is between once and twice its purge threshold, then counters accessed longer than 10 s (Windows 10) or 60 s (Windows Server) ago are considered stale; if the PathSet's size is more than twice its purge threshold, all counters are considered stale [45].

Linux. The Linux kernel (v6.9) uses either per-connection or per-bucket IPID selection depending on the connection type. Packets sent via established sockets use per-connection selection, initializing connection counters with a random value generated by `prandom_u32` after the connection is established. Atomic packets sent outside of sockets and any RST response to an unsolicited packet (e.g., a stray SYN/ACK) are assigned IPID zero [1]. All other packets use per-bucket selection with 2,048 to 262,144 (i.e., 2^{11} to 2^{18}) buckets, depending on the sender's RAM. A packet is hashed to a bucket using SipHash-2-4 on its destination IP address, source IP address, protocol number, and a 128-bit random key. Instead of incrementing bucket counters sequentially (i.e., by one, modulo 2^{16}), Linux uses *stochastic increments* based on the number of system ticks since a bucket counter was last incremented. Formally, if a bucket counter with value c was last accessed at time t_{old} and is being accessed for a new IPID at time t_{now} (where times are expressed in system ticks), then an increment *inc* is sampled uniformly at random from $[1, \max\{1, t_{now} - t_{old}\}]$ and the new IPID and bucket counter value is $c + inc \bmod 2^{16}$. These stochastic increments were designed to add noise to less frequently used bucket counters that adversaries could otherwise take advantage of.

OpenBSD. OpenBSD (v7.5) implements PRNG IPID selection using an iterated Knuth shuffle with $k = 32,768$ (i.e., 2^{15}). Random swap indices are generated using `arc4rand`. OpenBSD never assigns zero as an IPID: if zero is next in the permutation, it is swapped as usual but not returned.

FreeBSD. FreeBSD (v14.0) assigns IPID zero to all atomic packets and uses globally incrementing selection for non-atomic packets. FreeBSD includes a configuration option to use PRNG selection with a searchable queue of size $k = 8,192$ (i.e., 2^{13}) instead, though it is disabled by default to minimize performance impact. This disabled code uses `arc4rand` as its PRNG. As in OpenBSD, zero is treated as a special IPID that is never returned.

macOS/XNU. Although most source code for macOS is not publicly available, its network stack uses the open source XNU kernel (v8792.41.9). As in FreeBSD, macOS/XNU assigns zero as the IPID of all atomic packets. For non-atomic packets, macOS recently adopted pure PRNG selection, generating 16-bit values uniformly at random and then salting them with a packet-specific value; notably, this implementation does not include mechanisms for avoiding repeated IPIDs. As in the BSDs, zero is treated as a special IPID that is never returned.

3 MEASUREMENTS AND ATTACKS

In this section, we present a history of IPID-based exploits and the corresponding OS implementation changes they influenced (see Fig. 2 for an overview). Notably, many vulnerabilities remained exploitable long after disclosure due to delays in patching and updating to the latest versions. We categorize these exploits as either *probe comparisons* (Section 3.1) or *fragment injections* (Section 3.2) based on how they use IPIDs to achieve their goals. Within these categories, we distinguish between *measurements* which infer information about other machines and *attacks* which maliciously affect
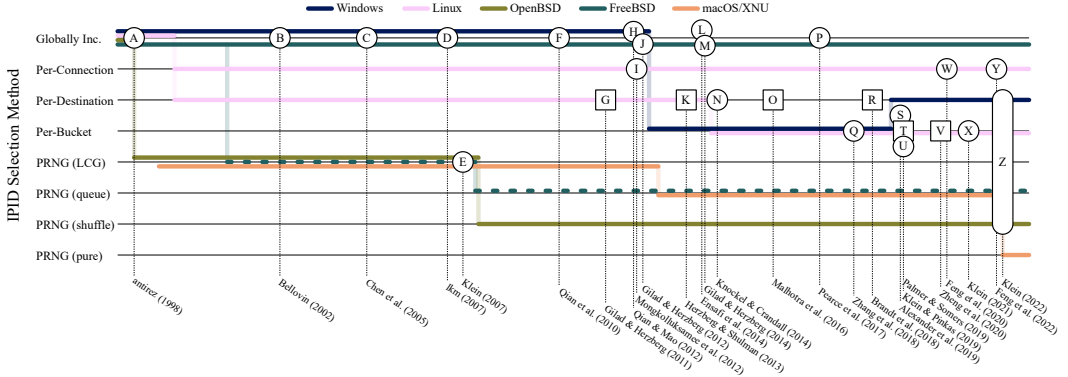
Fig. 2. *Timeline of IPID Measurements, Attacks, and Selection Methods.* Each OS implementation of IPID selection varies over time in response to disclosed measurements and attacks (Section 3), categorized either as probe comparisons (circles) or fragment injections (squares). Since 2000, Linux has implemented two methods, one for atomic packets (per-connection) and one for non-atomic packets (per-destination or per-bucket). FreeBSD has implemented various PRNG-based methods (dashed lines) but always disables them by default. Minor patches that did not change selection methods (e.g., a different hash function in per-bucket selection or Linux's fixes for specific packet types) are not shown.

target systems—often by leveraging information learned from measurements. We conclude with a brief discussion of related exploits against IPv6 fragmentation (Section 3.3).

Network exploits by a machine M against target machines A and B are performed from one of three vantage points: off-path, on-path, or in-path [54]. If M is *off-path*, M does not receive any traffic between A and B. Instead, M must actively *probe* these machines by sending them packets and inspecting their responses to learn anything about their states. An *on-path* machine passively receives all traffic between A and B but cannot modify it directly; an *in-path* machine (i.e., a "man-in-the-middle") receives and can modify all traffic between A and B. For example, if an ISP were to forward copies of its traffic to its government, the ISP is in-path and the government is on-path. We are primarily concerned with off-path exploits, since these rely on side channels like IPIDs for information. They are also the most powerful in practice, since they are usable by any unprivileged adversary without ISP or nation-state control over network infrastructure.

3.1 Probe Comparisons

A *probe comparison* tracks changes in IPIDs over time to infer information about other machines. As such, most probe comparisons are measurements. The inferred information is typically binary: whether a target port is open or closed, a firewall is performing egress or ingress filtering, a guessed secret value is correct or incorrect, and so on. Probe comparisons come in many forms, but the archetypal approach for an off-path measurer or attacker works as follows:

(1) Probe the target machine's relevant IPID, obtaining value x.
(2) Interact with the target machine such that its IPID values change one way if the inferred information is "true" (e.g., the target port is open, a firewall is performing egress filtering, a guessed secret value is correct, etc.) and some other way otherwise.
(3) Probe the target machine's relevant IPID again, obtaining a new value y.
(4) Based on the interactions in Step (2), compare x and y to gather evidence about whether the inferred information is "true". Repeat Steps (1)–(3) as needed to gain statistical confidence.

From this outline, we can characterize two necessary conditions for successful probe comparisons:

(PC1) A measurer or attacker must be able to probe and interact with a target machine's relevant IPID selection mechanism, either directly or indirectly, in a timely manner.

(PC2) A measurer or attacker must be able to correlate its interactions with the target machine to observable changes in the target machine's IPID values.

The remainder of this section summarizes seven probe comparison exploits grouped by their end goals, including NAT measurements (Section 3.1.1), idle scans (Section 3.1.2), triangular spamming (Section 3.1.3), communication measurements (Section 3.1.4), bucket leaks (Section 3.1.5), TCP injection attacks (Section 3.1.6), and covert channels (Section 3.1.7).

3.1.1 On-Path NAT Measurements. It was once assumed that *network address translation* (NAT) [69] obfuscated devices in a local network behind a shared public IP address, stopping external observers on the public network from counting or identifying NATted devices or distinguishing their traffic. However, early NAT devices did not change outgoing packets' IPIDs. Bellovin [6] (Fig. 2B) showed how this could be exploited for on-path measurement (e.g., by an ISP that sees all NATted traffic but does not manipulate it) if consecutive IPIDs assigned by the same NATted device are correlated. Notably, this on-path vantage point eliminates the need for probe comparison's probing and interaction steps, simplifying it to comparing IPIDs over time.

Bellovin's rules for matching IPID sequences to NATted devices are based on globally incrementing IPIDs, but could easily extend to any counter-based method by changing parameters. Thus, the paper's suggested mitigations included using a PRNG-based selection method. However, this protection alone proved insufficient: Mongkolluksamee et al. [57] (Fig. 2I) later extended this on-path measurement to uniquely identify NATted devices and their OSes across all IPID selection methods by additionally observing patterns in the TCP sequence number and source port.

3.1.2 Idle Scans. In the previous section, an on-path vantage point enabled IPID-based measurements that could be performed entirely passively. However, off-path measurements require the active transmission of packets, not just to probe current IPID values but often also to record the effect (or lack thereof) of a packet transmission on subsequent IPIDs. The earliest known example of such a measurement is the *idle scan* which exploits the predictability of globally incrementing IPIDs to reveal off-path connections [2, 18, 33] (Fig. 2A). An off-path measurer M can learn whether a target machine T has a certain port open by probing the IPIDs of an intermediate "zombie" machine Z with low traffic rates (Fig. 3). First, M sends Z an unsolicited SYN/ACK packet, to which Z replies with an RST packet that contains its current IPID, say x. Machine M then sends a spoofed SYN packet to T appearing to originate from Z destined for the target port it wants to scan. If this port is closed, nothing comes of this spoofed packet (Fig. 3a); otherwise, if it is open, T replies to Z with a SYN/ACK, which in turn causes Z to reply to T with an RST containing an incremented IPID (Fig. 3b). Finally, M again sends Z an unsolicited SYN/ACK and inspects the incremented IPID y of the RST reply. If $y \equiv x + 2 \bmod 2^{16}$, M concludes the target port on T is open; otherwise, if $y \equiv x + 1 \bmod 2^{16}$, it concludes the port is closed or filtered.

As a result of the idle scan's disclosure, OpenBSD, FreeBSD, and XNU replaced their globally incrementing IPIDs with PRNG selection using Linear Congruential Generators (LCGs) with additional complication layers. These implementations lasted until 2007 when Klein [42] (Fig. 2E) demonstrated that the "random" values produced by LCGs could be predicted, once again making idle scans possible. In response, OpenBSD replaced LCGs with an iterated Knuth shuffle (see OpenBSD commit 4fd19f) while FreeBSD—and later, XNU—chose PRNG selection using searchable queues (see FreeBSD commit 361021 and XNU commit ff3a0c). These implementations remain largely unchanged since their appearances in 2008. Linux took a different approach, implementing
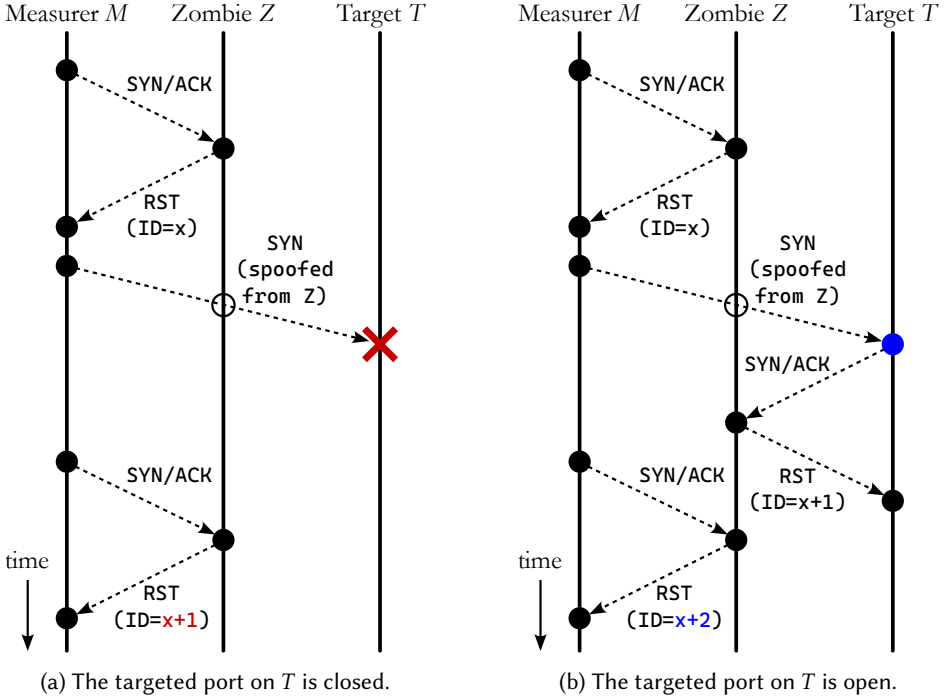
(a) The targeted port on T is closed.

(b) The targeted port on T is open.

Fig. 3. *Idle Scan Execution.* In an idle scan, a measurer M is testing whether a target port on another machine T is open. Measurer M first probes the zombie machine Z's current IPID, x. It then spoofs a SYN packet to the target machine T that appears to have come from Z. (a) If the target port on T is closed (red X), then T does not respond and M's final probe of Z's IPID is $x + 1 \mod 2^{16}$. (b) Otherwise, if the target port on T is open (blue circle), the resulting interaction between T and Z causes M's final probe of Z's IPID to be $x + 2 \mod 2^{16}$.

a hybrid of per-connection selection (for atomic packets) and per-destination selection (for non-atomic packets). Windows retained globally incrementing IPIDs until Windows 8, which was released in 2012 [59].

Despite these changes, delayed patching gave ample opportunity for the idle scan's wide adoption and refinement. Nmap integrated it as a scanning option in 2001 [52]. Chen et al. [17] (Fig. 2C) introduced an extension that additionally inferred packet loss, duplication, and arrival order; server traffic rates; and the number of servers in a load-balanced deployment. Later, Ensafi et al. [22] (Fig. 2L) improved the idle scan to detect off-path SYN/ACK and RST filtering between machines [23], which in turn was leveraged for Internet-wide censorship measurement [21, 24, 63] (Fig. 2P).

3.1.3 Triangular Spamming. Qian et al. [66] (Fig. 2F) formalized *triangular spamming*, a technique for email spam where a high-bandwidth spammer spoofs spam through low-bandwidth relay bots to a target mail server, thus concealing its own IP address and protecting itself from IP block-listing. IPID probe comparisons are less relevant to the actual execution of triangular spamming, but play a key role in its setup when the attacker identifies susceptible ISPs and relay bots. Specifically, the relay bot(s) must be able to receive responses inbound on port 25 from the mail server and forward them to the spammer using some other port; i.e., the ISP must not be ingress filtering on port 25.

If a potential relay bot R uses globally incrementing IPIDs, the presence of ingress filtering on port 25 can be determined using a simple probe comparison. The attacker A first probes R with
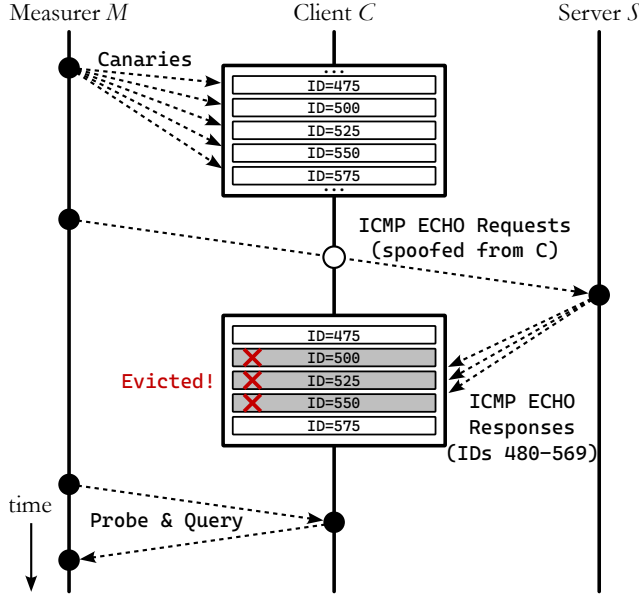
Fig. 4. *Canary Fragments Execution.* If a server S uses a sequentially incrementing counter for its IPIDs, a measurer M can count packets sent from S to a client C by planting canary fragments with a range of guessed IPIDs in the reassembly buffer of C and later querying whether they were evicted.

an unsolicited SYN/ACK on port 80 and records the IPID x of the RST reply. It then sends a large burst of $s \gg 1$ probes on port 25 which each increment the global IPID counter on R if and only if inbound traffic on port 25 is not blocked. Finally, A again probes R on port 80 to obtain its updated IPID y. If $y \approx x+1 \bmod 2^{16}$, the ISP is likely blocking inbound traffic; otherwise, if $y \approx x+s \bmod 2^{16}$, the relay bot is receiving packets on port 25 and can be used in triangular spamming.

3.1.4   Off-Path Measurement of Inter-Machine Communication. For 14 years, Linux's per-destination IPIDs for non-atomic packets seemed impervious to probe comparisons. If an off-path measurer M probes the per-destination counter on a server S for some client C by spoofing a packet from C to S, the server S replies to C, not M. But this intuition also reveals a weakness: every increment of a per-destination counter is caused by a packet sent to its unique destination. Knockel and Crandall [47] (Fig. 2N) recognized that this could be abused to detect if S and C are communicating and, if so, count the number of ICMP, UDP, or TCP packets exchanged between them. First, M fills the reassembly buffer of C with *canary fragments* containing guessed IPIDs that appear to originate from S (Fig. 4). Second, M spoofs large ICMP ECHO requests from C to S which become fragmented ICMP ECHO responses from S to C. If any canary has an IPID matching an ECHO response, it is evicted from the reassembly buffer. Finally, M "queries" the reassembly buffer of C to determine if many canaries are missing; if so, their guessed IPIDs were correct. By repeating this measurement over time and tracking the change in IPIDs, M can count the number of packets sent from S to C.

When canary fragments were disclosed, the Linux developers had already been working to replace per-destination selection—due to its large performance impact on modern servers—with per-bucket selection using a Jenkins hash of the destination IP address (see Linux commit 73f156). To curb canary fragments' abuse of per-destination counters' perfect increments-to-packets correlation, they also added the stochastic increments described in Section 2.3 and included the source IP address and protocol number in the bucket hash input (see Linux commit 04ca69).

3.1.5 Bucket Leaks. Conceptually, per-bucket selection blends globally incrementing selection's noisiness of shared counters with per-destination selection's difficulty of direct probing, but in practice has proven vulnerable to the side channels of both. Recall that a bucket hash function h hashes packets to buckets using a packet's source and destination IP addresses s and d, the protocol number p, and a secret key k. If a measurer controls an IP address d' such that $h(s, d, p, k) = h(s, d', p, k)$, it can exploit this *hash collision* to probe the per-bucket counter assigning IPIDs to machine d. Below, we summarize three measurements exploiting this side channel.

Non-Idle Port Scans. Zhang et al. [71] (Fig. 2Q) introduced the first method for discovering bucket hash collisions and used it to perform off-path port scans without an idle machine. To find a hash collision, a measurer M first sends an unsolicited SYN/ACK to a server S and records the RST reply's IPID, say x, which is drawn from bucket $b_M = h(S, M, p, k)$. Measurer M then spoofs a SYN/ACK to S that appears to originate from the client C that M is trying to collide with. Server S replies to C with an RST whose IPID is drawn from bucket $b_C = h(S, C, p, k)$. Finally, M again probes bucket b_M of S to obtain IPID y. Importantly, M sends all three SYN/ACKs in rapid succession to avoid the per-bucket counter's stochastic increments. If $y \equiv x + 2 \bmod 2^{16}$, M concludes that $b_M = b_C$; otherwise, if $y \equiv x + 1 \bmod 2^{16}$, M and C are likely hashing into different buckets.

On failing to find a hash collision, M can repeat this process with a different IP address it controls. The probability of finding a hash collision for one of B buckets when controlling D IP addresses is $1 - (1 - 1/B)^D$, which is over 63% when $D \approx B$ for the typical range of $B \in [2^{11}, 2^{18}]$. At the time this scan was discovered, similar collision probabilities were achievable with fewer IPv4 addresses when targeting dual-stack machines (those with both IPv4 and IPv6 addresses), since it is relatively easy to obtain a large block of IPv6 addresses and the same counters were used for both per-bucket IPIDs and IPv6 fragment identifiers. Now, Linux generates purely random, non-zero IPv6 fragment IDs (see Linux commit 62f20e), mitigating this dual-stack weakness and related issues [44].

Once a collision is found, the remainder of the port scan operates like an idle scan (Section 3.1.2) with added statistical inference to overcome per-bucket counters' stochastic increments.

Hybrid Leaks. Alexander et al. [1] (Fig. 2S) introduced *hybrid leaks*, a measurement that detects active TCP connections between a Linux server S and any of its clients C by differentiating between Linux's hybrid modes of per-bucket and per-connection selection. First, a measurer M finds a hash collision with C on S using a simplified version of the Zhang et al. [71] method described above. It then sends an unsolicited SYN/ACK to S and obtains the current per-bucket IPID x from the RST reply. Next, M spoofs $s > 1$ SYN/ACKs from C to S containing a guessed source port and a standard TCP destination port (i.e., 80 or 443). If S and C have an active TCP connection on those ports, S replies to C with a single "challenge ACK" whose IPID is drawn from their per-connection counter. Otherwise, S replies to C with s RST packets whose IPIDs each increment the per-bucket counter. Finally, M probes the updated per-bucket IPID y. Assuming M sends its packets fast enough to avoid the per-bucket counter's stochastic increments, $y - x \bmod 2^{16} < s$ is evidence that S and C have an active connection.

Linux mitigated this probe comparison measurement—and others depending on unsolicited SYN/ACK probes and their RST replies [2, 51, 66, 71]—by replying to any unsolicited SYN/ACK with an RST whose IPID is zero (see Linux commit 431280).

Device Fingerprinting. Klein and Pinkas [46] (Fig. 2U) leveraged a combination of reverse-engineering, bucket hash collisions, and cryptanalysis using sequences of IPID values to exfiltrate the secret keys Windows and Linux devices used in their bucket hash functions. Since these secret keys are generated randomly at device startup and remain fixed until a device restart, they form a long-lived device fingerprint, enabling a measurer to track devices across browsers, private sessions,

VPNs, and network configuration changes. At a high level, the Windows exfiltration technique takes advantage of two facts: (1) the hash function Windows used at the time assigns all destination IP addresses from the same class B network to the same bucket counter, and (2) the same secret keys and a related hash function are used to calculate offsets that are added to a counter values before returning them as IPIDs. This enables a measurer to probe a target bucket counter from a small number of IP addresses across carefully chosen class B networks and then substitute the returned IPIDs into a system of linear equations whose (very small) set of solutions necessarily contains a sufficiently large portion of the device's secret key to be used as a fingerprint. The Linux exfiltration technique leverages bucket collisions in a different way, probing a target machine in bursts from a range of IP addresses and then collecting pairs of IPIDs that are probabilistically likely to have originated from the same bucket. All possible secret keys are then exhaustively searched, and the one predicting the highest number of collision pairs is considered the true key.

In response to this exploit, Linux developers increased the secret key size from 64 to 128 bits and updated the hash function to SipHash-2-4, which remains the current implementation (see Linux commit df4537). Notably, Windows chose to address this issue by replacing per-bucket selection with per-destination [44] which, as already discussed in Section 3.1.4, has a high performance impact and has been exploited successfully before [47, 53].

3.1.6   TCP Injection Attacks. Beyond measuring information that an off-path observer would not otherwise be privy to, probe comparisons can also lay the groundwork for attacks that control or affect resources typically assumed to be beyond an off-path attacker's influence. In a *TCP injection attack*, an attacker inserts malicious traffic into an active TCP connection between a client and server. To do so, the attacker needs to know the client and server's IP addresses and TCP ports along with the current TCP sequence number of the machine the attacker is spoofing. Most attacks assume the two IP addresses are known and the server is using a standard port for TCP (e.g., 80 or 443), leaving the client's port and sequence number as the two unknowns. But if the client uses globally incrementing IPIDs, an off-path attacker can learn the client's port and sequence number through a port comparison guess-and-check [51] (Fig. 2D). The attacker (1) probes the client's IPID, say x, (2) sends packets that are specifically crafted to make the client increment its IPID if and only if its guessed port or sequence number is correct, (3) probes the client's new IPID, say y, and (4) compares x and y to determine the accuracy of its guess. Just like for idle scans (Section 3.1.2), this attack's second step depends critically on the connection having little traffic, since other packets incrementing the client's IPID would confuse the results. However, Gilad and Herzberg [29, 31] (Fig. 2J,M) showed that with a "puppet" on the client (e.g., a malicious script in a browser sandbox), this attack can succeed even on noisier connections.

Naturally, the defense against this guess-and-check technique and the resulting TCP injections is to use an IPID selection method that is less predictable or harder for an attacker to access. However, this must be done on all machines involved in the connection, not just the client. Qian and Mao [65] (Fig. 2H) showed that even if a client's IPIDs are difficult to predict, a firewall or middlebox using globally incrementing IPIDs can leak the sequence numbers necessary for TCP injection.

The state-of-the-art attacks remained specific to globally incrementing IPIDs for 13 years, until Feng et al. [25–27] (Fig. 2W,Y) introduced their novel *downgrade attacks* to achieve TCP injection against Linux's hybrid per-connection/per-bucket IPIDs. Recall that when guess-and-checking the client's TCP port and sequence numbers, the attacker needs to probe the client's relevant IPIDs. But it is impossible for the attacker to probe the per-connection counter specific to this client–server's active TCP connection unless it has the very information it's trying to leak. Instead, Feng et al. observe that per-connection selection can be downgraded to per-bucket selection by spoofing ICMP "Fragmentation Needed" packets from the client to the server, tricking it into clearing its responses'

DF flags and thus demoting otherwise atomic packets to non-atomic. The client's corresponding bucket counter can then be identified and probed using the hash collision techniques described in Section 3.1.5, and an appropriately modified version of the above guess-and-check technique will leak the client port and sequence number(s) needed for TCP injection. Feng et al. demonstrated that this method can be used to poison HTTP and BGP traffic in the wild [27]. Linux mitigated this issue in a case-specific manner, replacing per-bucket selection with an IPID generated by `prandom_u32` only for SYN/ACKs that are large enough to be fragmented (see Linux commit 970a5a).

3.1.7 Covert Channels. A *covert channel* is an attack that transfers data between two processes or machines that are not supposed to be able to communicate, e.g., a privileged program leaking data to an unprivileged user [49]. Klein [44] (Fig. 2Z) demonstrated that both counter-based and PRNG-based IPIDs can be used as covert channels to transmit information from a sender within an isolated network to any external receiver. Although the details differ across OSes and IPID selection methods, the main idea is for the isolated sender to force its outside-facing firewall or host machine to either send a number of packets (signalling a 1 bit) or not (signalling a 0 bit), which the external receiver can view via probe comparison of the firewall/host's IPIDs.

Perhaps the most surprising of these covert channels uses PRNG selection with a searchable queue of size k, since PRNG IPIDs' unpredictability is not a defense in this case. First, the receiver quickly sends k probes to the firewall/host so that the replies' IPIDs $x_1, \ldots, x_k$ are exactly those in the searchable queue. The isolated sender then communicates a 0 or 1 bit as described above. Finally, the receiver sends another k probes to the firewall/host to obtain IPIDs $y_1, \ldots, y_k$. If there is overlap between the x_i's and y_i's, some packets must have been sent between the two sets of probes to evict the original members in the queue, likely indicating that the sender communicated a 1 bit; otherwise, if the x_i's and y_i's are disjoint, the sender likely communicated a 0 bit.

All major OSes except Windows and OpenBSD released patches for these covert channels. In particular, macOS mitigated this issue by removing the searchable queue from its PRNG selection, abandoning the idea of reserving IPIDs for non-repetition altogether (see XNU v8019.61.5, which introduced the change, and v8792.41.9, which fixed a minor bug for avoiding IPID zero).

3.2 Fragment Injections

In a *fragment injection*, the attacker plants one or more malicious fragments appearing to originate from some trusted source in a target machine's reassembly buffer. If any of these fragments' IPIDs match that of a legitimate fragmented packet from the trusted source, a malicious fragment may replace a legitimate one during reassembly. This technique has been used to poison DNS caches and forge domain validation certificates (Section 3.2.1), intercept traffic behind a NAT (Section 3.2.2), and shift time on NTP clients (Section 3.2.3). The general template for fragment injection is:

(1) Ensure traffic from the trusted source to the target machine will be fragmented (e.g., by spoofing an ICMP "Fragmentation Needed" packet from the target to the source).
(2) Predict the IPID(s) that the trusted source will assign to the packets targeted for fragment injection, potentially using the probe comparisons outlined in Section 3.1.
(3) If needed, use path MTU discovery or other measurement methods to learn the necessary parameters for crafting malicious fragments.
(4) Spoof malicious fragments from the trusted source to the target machine.

There is one salient necessary condition for successful fragment injections:

(FI1) An attacker must be able to guess, with reasonable probability, the IPID(s) that the trusted source will assign to the packets targeted for fragment injection.
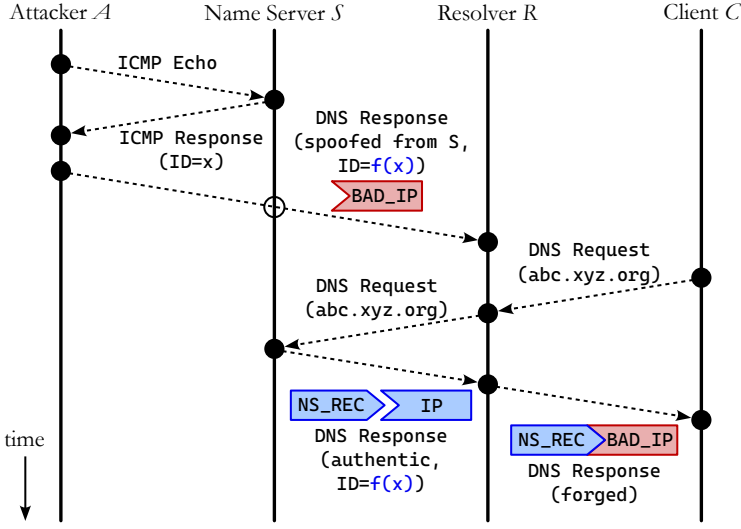
Fig. 5. *DNS Cache Poisoning Execution.* In this attack, an attacker *A* is redirecting traffic for a specific domain (e.g., abc.xyz.org) to an IP address of its choosing. First, *A* probes the DNS name server *S*'s current IPID, *x*. It then spoofs a DNS response fragment containing its chosen IP address to the DNS resolver *R* that appears to have originated from *S*; critically, this fragment's IPID must match whatever IPID $f(x)$ that *S* will use in its later DNS response. When *S* responds to a DNS request for abc.xyz.org, its fragment containing the authentic IP address will be discarded in favor of the spoofed fragment with the same IPID already in *R*'s reassembly buffer, causing the reassembled DNS response to contain the attacker's chosen IP address.

Fragment injections can be viewed as a special case of *(de)fragmentation attacks*, exploits that abuse a machine's reassembly buffer size, fragment eviction policy, or reassembly strategy for unintended behavior. For example, a denial of service attack known as "teardrop" abused the fact that older reassembly methods would fail if fragments overlapped by a few bits (CVE-1999-0015). Similar attacks were successful across every major OS at some point in time (e.g., CVE-1999-0052, CVE-1999-0157, CVE-1999-0431, CVE-2000-0305, and CVE-2004-0744), and modern fragmentation attacks continue to be a concern for IP stack implementations (e.g., CVE-2020-3373, CVE-2020-28041, CVE-2021-3905, and CVE-2023-24821). We focus specifically on fragment injections since these depend critically on IPID selection methods and their predictability.

3.2.1 DNS Attacks. In *DNS cache poisoning*, an attacker redirects traffic to an IP address of its choosing by replacing a DNS resolver's cache entry for a valid domain name [67]. The first cache poisoning attacks exploited the predictability of globally incrementing DNS transaction IDs (TXIDs) in the BIND protocol to trick DNS servers into accepting spoofed DNS replies [3]. But forging entire DNS replies is not necessary for poisoning. Instead, Herzberg and Shulman [34, 35] (Fig. 2K) showed that poisoning can be achieved by placing spoofed fragments of a DNS response in a DNS resolver's reassembly buffer. If any of these fragments' IPIDs match that of the real DNS response—which is easily guessed for globally incrementing or per-destination selection—a spoofed fragment will be reassembled with the rest of the DNS response, poisoning the cache (see Fig. 5 for details).

Palmer and Somers [61] (Fig. 2T) later refined DNS cache poisoning to succeed on its "first try"—i.e., with a single spoofed fragment—and Zheng et al. [73] (Fig. 2V) adapted it to work against DNS forwarders. Klein [43] (Fig. 2X) revealed a weakness in the PRNG that Linux used to initialize its per-connection counters and generate per-bucket's stochastic increments, making DNS cache

poisoning possible with per-connection and per-bucket IPIDs as well (see Linux commit f227e3). Finally, beyond using DNS cache poisoning to redirect traffic, Brandt et al. [11] (Fig. 2R) poisoned certificate authorities to obtain fraudulent certificates for arbitrary domains.

3.2.2 NAT Packet Interception. Gilad and Herzberg [28, 30] (Fig. 2G) observed that when an external sender S using per-destination IPIDs communicates with a NATted receiver, it treats the NAT device as its destination and thus draws from the same per-destination counter for all machines behind the same NAT. Thus, if an off-path attacker A controls one of the NATted machines M, it is easy to obtain the sender's per-destination IPIDs: M simply pings S and reports the reply's IPID to A. Attacker A also needs the destination port that the NAT uses to map traffic from S to M, which A obtains by spoofing packets from S to the NAT with different guessed destination ports until one of them gets to M, who reports the correct destination port back to A.

Attacker A can use these exfiltrated parameters to intercept traffic sent from S to any receiver R behind the NAT. Suppose a legitimate packet sent by S is fragmented as (f_1, f_2), where f_1 contains the destination port for R and f_2 contains the payload. Using the predicted IPID, A plants a spoofed second fragment f_2' that is designed to reassemble with and discard the legitimate f_1 when it arrives. It then spoofs a new first fragment f_1' containing the destination port for M, the attacker-controlled NATted machine. When the legitimate f_2 arrives, the NAT reassembles it with f_1' and delivers it to M, completing the interception. Generalized versions of this attack achieve off-path denials of service against inter-gateway tunnels when the attacker has puppet scripts behind both endpoints.

3.2.3 Shifting NTP Time. Malhotra et al. [53] (Fig. 2O) discovered an off-path attack leveraging predictable IPIDs to inject traffic into Network Time Protocol (NTP) client–server connections and shift time on NTP clients. An attacker first spoofs an ICMP "Fragmentation Needed" packet from the target NTP client to its NTP server. It then uses the puppet technique of Gilad and Herzberg [28, 30] or the canary fragments technique of Knockel and Crandall [47] to accurately predict the server's per-destination IPIDs. This allows the attacker to spoof overlapping fragments from the server to the client containing phony timestamps which—depending on the client's reassembly policy for overlapping fragments—may get reassembled into the final packet. This attack was never explicitly patched, though Malhotra et al. report that only $\approx 0.2\%$ of NTP servers and $\approx 1.3\%$ of tested NTP clients were vulnerable in 2016. These numbers have likely only decreased as globally incrementing and per-destination IPIDs become less common.

3.3 Measurements and Attacks Using IPv6 Fragmentation

Although our primary focus in this survey is IPv4's IPID, related vulnerabilities exist for IPv6. Recall from Section 2.1 that fragmentation in IPv6 is performed only by senders and is achieved using the optional *Fragment* extension header which includes a 32-bit identifier (i.e., the IPv6 IPID) for disambiguating packets during reassembly. It was originally believed that IPv6 would obviate IPID-based exploits since (1) IPv6 packets only contain the *Fragment* extension header and its constituent IPID when fragmentation is needed and (2) a combination of the IPv6 minimum link MTU and path MTU discovery would enable senders to resize their packets and avoid fragmentation a priori. Unfortunately, as was the case for IPv4, path MTU discovery "is not panacea" [27]. Off-path measurers or attackers can use a carefully crafted combination of ICMPv6 "Echo Request" and "Packet Too Big" messages specifying an MTU smaller than the IPv6 minimum to force IPv6 fragmentation [58]. With this mechanism, sequences of IPv6 IPIDs can be probed and predicted to achieve off-path measurements and attacks analogous to those described in Sections 3.1 and 3.2; some notable examples include TCP idle scans [58], IPv6 alias resolution [7], and IPv6 router uptime characterization [8]. A recent study probing 20 million IPv6 addresses reports that 46.1% of probed machines yield predictable IPv6 IPID sequences and may be vulnerable to IPID-based exploits [37].

4 COMPARATIVE ANALYSIS

Section 3 chronicled the co-evolution of IPID selection methods' OS implementations and the diverse measurements and attacks that exploited them. In this section, we present a unifying mathematical model of IPID selection (Section 4.1) and use it to compare and evaluate the methods' *correctness* (Section 4.2), *security* (Section 4.3), and *performance* (Section 4.4). By parameterizing this comparison by a machine's expected rate of traffic, we reveal each selection method's relative (dis)advantages across a spectrum of use cases, summarized in Table 1. These evaluations form the basis of our discussion and recommendations in Section 5.

Several works surveyed in Section 3 include formal modeling of IPID selection [17, 22, 34, 44, 71], but focus only on the exploit or target selection method at hand. Similar modeling efforts exist for TCP sequence numbers, where security is analyzed only with respect to specific exploits (e.g., [14, 56, 65, 72]) and performance analyses focus primarily on congestion control (e.g., [15, 55, 60]). Our goal, in contrast, is to evaluate all seven IPID selection methods on level footing, providing an intuitive synthesis for practitioners weighing the tradeoffs of different methods.

4.1 Mathematical Model

We model a server S communicating with one or more clients and analyze properties of the IPIDs it assigns to outgoing packets. To capture the stochastic nature of network traffic, we assume S sends packets with non-trivial IPIDs according to a Poisson process with rate $\lambda > 0$. We define the "unit time" of this Poisson process as the average time for a packet sent by S to be routed to its destination and have all its fragments either reassembled or evicted from the destination's reassembly buffer.[1] This time interval is important for two reasons. First, it is the window of uncertainty for adversaries trying to correlate IPIDs obtained by probing S (Section 3.1), since other packets that S sends in the unit time between its reply to some probe and when that reply is received will further change its current IPID value. Second, this is the interval during which fragments in the same reassembly buffer with the same IPIDs may interact, as required for fragment injections (Section 3.2).

Poisson processes are classically used in network traffic models that prioritize analytical elegance over high-fidelity representation and prediction. Specifically, packet interarrival times are adequately captured by simple Poisson processes when aggregating large-scale traffic [13] or over short time scales [40, 62], though traffic patterns do exhibit long-range dependence, self-similarity, and other non-stochastic properties [39, 50]. Alternative traffic models such as Markov Modulated Poisson Processes [4, 9] and autoregression models [16, 36, 62] can better capture these time-dependent properties, but are less amenable to the unifying comparative analysis we perform.

4.2 Analyzing Correctness

The intended function of IPIDs is to support unambiguous packet reassembly after IP fragmentation (see Section 2.1), so any measure of an IPID selection method's correctness should capture its ability to assign distinct IPIDs to packets whose fragments would otherwise be indistinguishable—i.e., those with the same source and destination IP addresses, source and destination ports (if applicable), and protocol number. Formally, we say the server S produces a *collision* at a client C if two fragments in the reassembly buffer of C belong to different packets sent by S but have the same IPID and protocol number. We analyze each selection method's correctness in terms of its *worst-case probability of producing a collision*, establishing a basis for fair comparison across both counter-based and PRNG-based methods.

---

[1]In practice, unit time can range from ~10 ms (a fast ping) to 120 s (the longest recommended reassembly timeout [10, 64]). Assuming a unit time of 10 ms and an average packet size of 1,500 bytes, 1 Kbps translates to $\lambda \approx 2^{-10.2}$, 1 Mbps translates to $\lambda \approx 2^{-0.3}$, 1 Gbps translates to $\lambda \approx 2^{9.7}$, 100 Gbps translates to $\lambda \approx 2^{16.3}$, and 1 Tbps translates to $\lambda \approx 2^{19.7}$.

Table 1. *Summary of IPID Selection Method Comparisons.* The key comparisons and takeaways for each IPID selection method analyzed in Section 4. For brevity, we use ✅ to indicate correctness, 🔒 for security, ⏱ for time complexity, and 🗄 for space complexity.

Selection Method	Strengths	Weaknesses
Globally Incrementing	Optimal ✅. Good 🔒 for sufficiently fast traffic, since rapid increments weaken correlation of probed IPIDs. Excellent ⏱ and 🗄, as atomic increments of a single 16-bit counter are fast and boast superior scalability under CPU contention.	Poor 🔒 for slow and moderate traffic, as infrequent, sequentially-incrementing counter values are very easily predicted.
Per-Connection	Optimal ✅, 🔒, and ⏱, tying with or outperforming all other methods, though "optimal 🔒" assumes off-path adversaries cannot probe connection counters.	Only usable by connection-bound traffic (incompatible with UDP, QUIC, etc.). Susceptible to downgrade attacks shifting to other methods. Unbounded 🗄.
Per-Destination	Optimal ✅. Reasonable 🗄, obtained at the cost of extra mechanisms ensuring the number of destination counters does not grow too large (see weaknesses).	The worst 🔒 and ⏱ of all methods. Even if overall traffic is fast, traffic for any one destination may be slow, making its seqentially-incrementing counter easy to predict. Managing stale counters is time-intensive, and CPU contention over the destination hash table hurts scalability.
Per-Bucket (Linux)	Optimal ✅, despite concerns about stochastic increments driving more frequent collisions. Near-optimal 🔒 for slow traffic, where large, noisy stochastic increments make prediction difficult. ⏱ Good scalability, though never as performant as globally incrementing.	Poor 🔒 for moderate and fast traffic, where counters are accessed so often that stochastic increments reduce to sequential increments, suffering similar predictability to per-destination. Reasonable 🗄 that scales with user RAM, but often heavier than other methods.
PRNG (searchable queue or iterated Knuth shuffle)	Worse ✅ than counter-based methods, but reserving sufficiently many IPIDs for nonrepetition mitigates birthday paradox collision issues for all but the fastest traffic. Near-optimal 🔒. Reasonable 🗄.	Reasonable ⏱ for single-CPU contexts, but scales very poorly due to contention over the searchable queue or cyclic permutation. Best for slow traffic settings.
PRNG (pure)	Optimal 🔒, producing IPIDs uniformly at random. ⏱ Fastest general-purpose method with trivial scalability (no CPU contention). 🗄 Memoryless.	Collision probabilities follow the birthday paradox, yielding the worst ✅ of any method. Usable only for slow traffic.

In the worst case, all packets sent by S are sent to the same client C via the same protocol and are fragmented in transit; thus, every packet has the potential to cause a collision. Recall from Section 4.1 that we assume outgoing traffic from S occurs according to a Poisson process with rate $\lambda > 0$. Let N be the corresponding random variable counting the number of packets "simultaneously in transit" from S to C, i.e., those sent by S that have not yet been reassembled or evicted by C. Then N is Poisson-distributed, characterized by probability mass function $\mathrm{pmf}(N, \lambda) = \lambda^N e^{-\lambda}/N!$ and expected value $\mathrm{E}[N] = \lambda$. Formally, the worst-case probability of S producing a collision at C
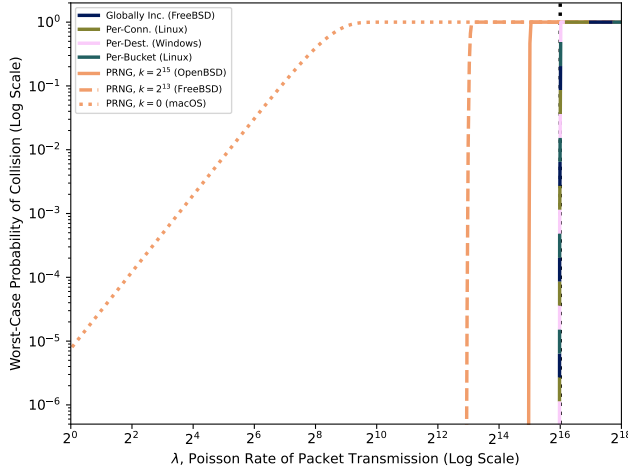
Fig. 6. *Comparison of IPID Selection Methods' Correctness.* Worst-case collision probabilities for globally incrementing (dark blue, Eq. 8), per-connection (olive, also Eq. 8), per-destination (pink, also Eq. 8), per-bucket (teal, simulation), and PRNG-based (orange, Eq. 10) IPID selection as a function of λ, the Poisson rate of packet transmission. The total number of distinct IPIDs, 2^{16}, is shown as a black dotted line. Note that the collision probabilities of the four counter-based methods coincide.

as a function of λ is

$$\Pr\left[\text{collision}\right] = \sum_{n=1}^{\infty} \Pr\left[\text{collision} \mid N = n\right] \cdot \Pr\left[N = n\right]$$

$$= \sum_{n=1}^{\infty} \Pr\left[\text{collision} \mid N = n\right] \cdot \text{pmf}(n, \lambda), \tag{1}$$

where $\Pr\left[\text{collision} \mid N = n\right]$ is the probability that any of the n distinct packets simultaneously in transit from S to C are assigned the same IPID (which varies by IPID selection method) and $\Pr\left[N = n\right] = \text{pmf}(n, \lambda)$ follows from N being Poisson-distributed.

In Appendix A.1, we derive these worst-case collision probabilities for all seven IPID selection methods. The results are shown in Fig. 6. Globally incrementing, per-connection, and per-destination selection all increment sequentially, so their collision probabilities are effectively zero until there are so many packets simultaneously in transit that it is probable to exhaust all 2^{16} IPIDs and reuse at least one. Perhaps more surprising is that per-bucket selection has nearly identical collision probabilities to the other counter-based methods even though its stochastic increments may skip some IPID values, cycle back around, and cause a collision in fewer than 2^{16} packets. To understand why, recall that a per-bucket counter's increment value is chosen uniformly at random from 1 to the number of system ticks since the counter was last incremented (Section 2.3). Any traffic rate λ that generates enough packets to cause a probable collision also requests IPIDs so frequently that the per-bucket counter effectively increments sequentially.

PRNG-based methods yield non-negligible collision probabilities at slower traffic rates λ than the counter-based methods do, with larger numbers k of IPIDs stored for non-repetition corresponding to better collision avoidance. Pure PRNG selection ($k = 0$) yields particularly poor correctness, succumbing to the birthday paradox: when n packets ("people") are assigned IPIDs uniformly at random from among the 2^{16} possible values ("birthdays"), it becomes quite likely that multiple

packets have the same IPID. At $\lambda = 2^5$ ($\approx$ 40 Mbps), there is a 1% chance of collision; at $\lambda = 2^7$ ($\approx$ 150 Mbps), this probability increases to 10%. Methods that reserve $k \gg 0$ IPIDs for non-repetition achieve negligible collision probabilities until transmission rates become relatively fast ($\lambda \geq k$).

4.3 Analyzing Security

Section 3 categorized numerous IPID-based exploits as probe comparisons and fragment injections. As highlighted by Conditions PC2 and FI1, the fundamental issue of *predictability* in IPID generation underlies the success of both types of exploits. With this motivation, we analyze each selection method's security in terms of its *probability of an adversarial guess* which, informally, is the ability of an off-path adversary A to guess the next IPID a server S will assign in real time.

Formally, consider an off-path adversary A trying to predict the next IPID generated by a particular IPID resource $i \in \{1, \ldots, r\}$ (i.e., a counter or PRNG) on S. We assume A knows—but cannot influence—both the rate $\lambda > 0$ of all outgoing traffic from S and the rates λ_i of only those packets whose IPIDs are assigned by resource i. For globally incrementing and PRNG selection in which $r = 1$, we have $\lambda = \lambda_i$; for the selection methods with $r \geq 1$ counters, we have $\lambda \geq \lambda_i \geq 0$. We further assume that A can probe any IPID resource for its current IPID value except a per-connection counter, as no known exploits have demonstrated that this is possible for an off-path adversary.[2]

We allow A a budget of $g \geq 1$ guesses and assume A always guesses the g maximum-likelihood IPIDs given the information available to it. Thus, the probability of a successful adversarial guess is

$$\Pr[\text{adv. guess}] = \max_{G \subset [2^{16}]\,:\,|G|=g} \Pr[\text{next IPID} \in G] = \max_{G \subset [2^{16}]\,:\,|G|=g} \left\{ \sum_{x \in G} \Pr[\text{next IPID} = x] \right\}, \quad (2)$$

where $[2^{16}] = \{0, \ldots, 2^{16} - 1\}$ denotes the set of all possible IPIDs, G is a set of g distinct IPIDs, and $\Pr[\text{next IPID} = x]$ is the probability that x is the next IPID generated by resource i (which varies among IPID selection methods).

In Appendix A.2, we derive adversarial guess probabilities for the seven IPID selection methods as a function of λ_i. To facilitate a fair comparison of results across selection methods, we visualize the results in two ways. We first consider the "uniform case" where the λ-rate traffic is evenly distributed over all r IPID resources of S, yielding $\lambda_i = \lambda/r$ (Fig. 7, left). For per-connection, per-destination, and per-bucket selection where $r \geq 1$, this represents a baseline setting where all active connections or destinations generate roughly the same amount of traffic or where the bucket hash function distributes a similar amount of traffic to each bucket. We then consider the "worst case" where λ_i takes on whatever value is both feasible with respect to the total rate λ and maximizes the adversarial guess probability (Fig. 7, right), i.e.,

$$\lambda_i = \begin{cases} \lambda & \text{if } r = 1; \\ \arg\max_{0 \leq \lambda_i \leq \lambda} \{\Pr[\text{adv. guess}]\} & \text{if } r > 1. \end{cases} \quad (3)$$

These worst case λ_i's capture the many ways λ total traffic can be allocated among r different IPID resources (e.g., bucket counters), surfacing the allocation that makes a given resource maximally predictable. For example, is it easier to predict a given bucket counter b when b is responsible for nearly all λ traffic, or when b assigns very few IPIDs with the rest of the traffic being handled by other buckets? Together, the uniform case emphasizes how the number of IPID resources impacts security while the worst case captures the fundamental predictability of any one IPID resource.

---

[2]If ever some future exploit enables an off-path adversary A to probe per-connection counters, then the security of per-connection selection reduces to that of per-destination selection, as they both maintain multiple sequentially-incrementing counters (see Appendix A.2.2 for details). Alternatively, if a downgrade attack [25–27] forces S to use some method other than per-connection selection, then its security reduces to that of the downgraded method.
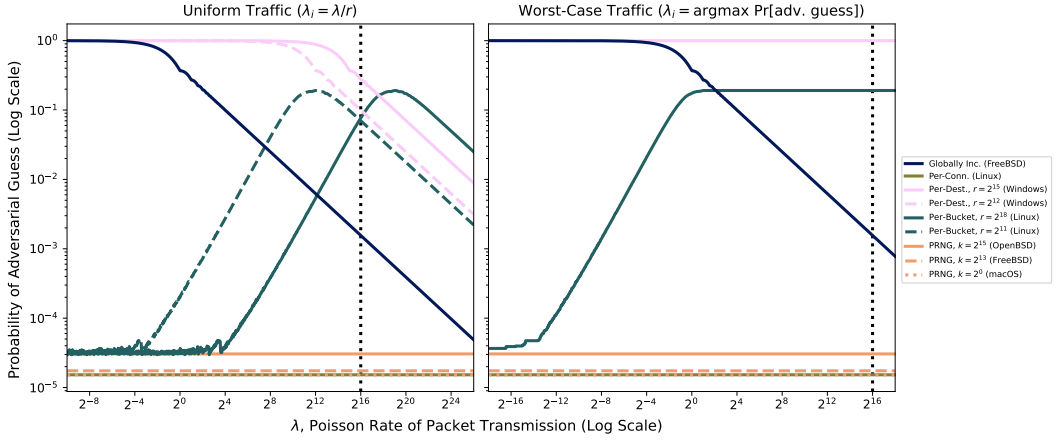
Fig. 7. *Comparison of IPID Selection Methods' Security.* Adversarial guess probabilities with $g = 1$ guess for globally incrementing (dark blue, Eq. 14), per-connection (olive, Eq. 15), per-destination (pink, also Eq. 14), per-bucket (teal, Eq. 17 with simulation), and PRNG-based (orange, Eq. 19) IPID selection as a function of λ, the Poisson rate of packet transmission. For selection methods that use $r \geq 1$ counters, we compare the case where traffic is uniformly distributed to each counter i with rate $\lambda_i = \lambda/r$ (left) and the worst-case scenario when traffic is allocated to λ_i such that the adversarial guess probability is maximized (right). The total number of distinct IPIDs, 2^{16}, is shown as a black dotted line. An extended version of this figure showing $g = 10$ and $g = 100$ adversarial guesses is given in Appendix A.2, Fig. 10. The absolute adversarial guess probabilities increase for larger g, but the comparisons among methods remain the same.

An IPID selection method is *optimally secure* if its probability of an adversarial guess is exactly $g/2^{16}$, i.e., if the adversary can do no better than make its g distinct guesses uniformly at random over all possible IPIDs. Per-connection and pure PRNG selection are thus the only optimally secure methods, though per-connection's optimality depends on the potentially precarious assumption that an off-path adversary can never infer the status of another machine's connection counter or downgrade per-connection selection to some less secure method. In contrast, the PRNG-based methods achieve (near-)optimal security based only on the assumption of a sufficiently random, correctly implemented, and well-seeded PRNG algorithm.

Our model confirms the probable success of idle scan attacks [2, 22], predicting near-certain adversarial guesses for globally incrementing selection on any quiet channel (Fig. 7, $0 < \lambda < 1$). It also confirms that Linux's addition of stochastic increments to per-bucket selection obfuscates otherwise predictable counter statuses in very quiet buckets ($\lambda < 2^{-12}$ for worst-case traffic; $\lambda < 2^{-1}$ for uniform traffic). However, this obfuscation quickly dissipates as the rate of traffic increases. Surprisingly, per-bucket selection is easier to predict than globally incrementing selection even at relatively low traffic rates ($\lambda > 2^2$ for worst-case traffic; $\lambda > 2^7$ for uniform traffic), revealing a subtle but significant security advantage to having all IPIDs assigned by a single resource. As traffic rates increase, the number of increments to the single, global counter between any two of the adversary's probes also increases, weakening the correlation between subsequent probed IPIDs; at very fast rates of traffic ($\lambda > 2^{16}$), this correlation is so weak that globally incrementing selection approaches the unpredictability of PRNG-based methods. Per-destination selection inherits the worst of both globally incrementing and per-bucket selection: it has multiple counters that always increment by one, rendering it the most predictable of all methods for all but the most extreme
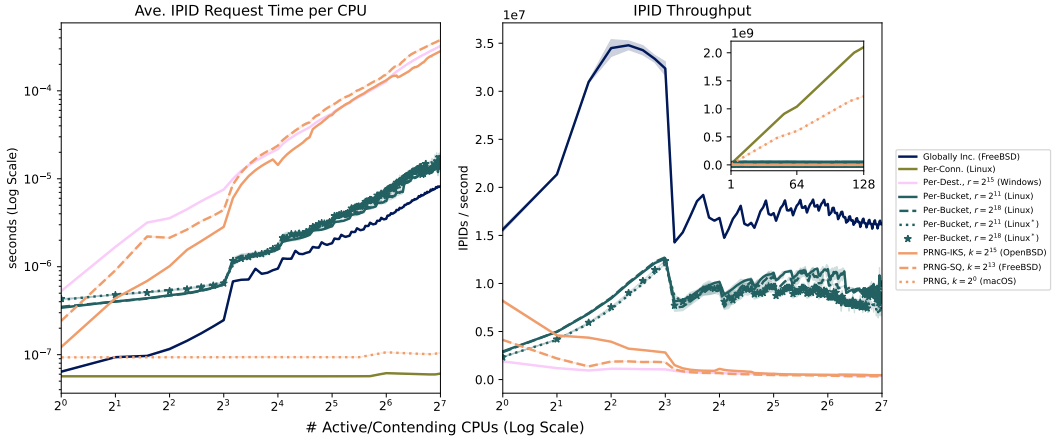
Fig. 8. *IPID Selection Methods' Time Complexities.* Average IPID request time per CPU (left) and IPID through-
put per second for all CPUs combined (right, right inset) as a function of the number of CPUs concurrently
assigning IPIDs via globally incrementing (dark blue), per-connection (olive), per-destination (pink), per-
bucket (teal), and PRNG-based (orange) IPID selection. Per-bucket is implemented in two ways: those marked
"Linux" (teal, solid and dashed) mimic Linux's actual implementation where bucket counters and last access
timestamps are individually atomic but allow race conditions between them, while those marked "Linux*"
(teal, dotted and starred) maintain per-bucket locks. Values are shown as means (lines) and standard deviations
(error tubes) over ten independent trials per IPID selection method and number of CPUs.

traffic rates. Thus, like per-connection selection, its "security" lies not in its resilience to prediction
but in the difficulty of accessing its counters, which has been subverted before.

4.4 Analyzing Performance

Our final dimension of comparison is performance, measured in both time and space complexity.
We perform this analysis in the context of multi-core servers where CPUs may contend over
IPID resources. To assess time complexity, we benchmarked each IPID selection method in ten
independent trials per number of CPUs $c \in \{1, 2, \ldots, 128\}$ on a 128-core (dual-socket AMD EPYC
7713 Zen3), 512 GiB node on Arizona State University's Sol supercomputer [38].[3] In each trial,
we count the number of IPIDs each of the c CPUs assigns to packets in a 10 second period when
concurrently and repeatedly scanning over a CAIDA trace of ~29 million packets [12]. Details of
our IPID selection method implementations are given in Appendix B.1.

The average IPID request times and packet throughput rates for each method are shown in Fig. 8.
Unsurprisingly, per-connection selection is very fast (less than 0.1 µs per IPID request regardless
of the number of CPUs). When Linux gets IPIDs for connection-bound packets, it operates in the
context of a socket data structure that at most one CPU is accessing per time. Thus, obtaining an
IPID is as simple as incrementing the socket's counter without any need to locate the counter via
hashing or manage concurrency control. Pure PRNG selection is similarly fast regardless of scale,
since each CPU gets its own PRNG context. The other methods require concurrency control to
protect access to their shared IPID resources, causing their IPID request times to increase with the
number of contending CPUs. Among these methods, globally incrementing selection (implemented
as a single atomic counter) is the fastest, even when all 128 CPUs contend over it. Surprisingly, it

---

[3]We repeated this benchmark on a 64-core (single-socket Intel Xeon Silver 4216), 256 GiB machine to validate our results
are not architecture-dependent and obtained consistent results. See Appendix B.2, Fig. 11 for details.

Table 2. *IPID Selection Methods' Space Complexities.* Where relevant, $r \geq 1$ is the number of IPID resources (i.e., counters or PRNGs), and k is the number of IPIDs that PRNG-based methods reserve for non-repetition. *The Windows implementation of per-destination selection technically has unbounded space complexity, but this table lists the expected memory usage for Windows Server v1904 (see text).

Selection Method	#16-bit Counters/IPIDs	Other Data	Memory
Globally Incrementing	1	N/A	2 bytes
Per-Connection	$r \geq 1$ active connections	N/A	Unbounded
Per-Destination (Windows)	$r \leq 2^{15}$ active destinations*	r 32-bit times	$\leq$ 192 KiB*
Per-Bucket (Linux)	$r \in [2^{11}, 2^{18}]$ buckets	r 32-bit times	12 KiB–1.5 MiB
PRNG (searchable queue)	$k \in [2^{12}, 2^{15}]$ reserved IPIDs	2^{16} lookup bits	16–72 KiB
PRNG (iterated shuffle)	2^{16} IPIDs in shuffle	one 16-bit index	$\approx$ 128 KiB
PRNG (pure)	N/A	N/A	None

outperforms even the per-bucket implementations whose primary reason for maintaining numerous buckets is to improve performance by reducing contention. Globally incrementing selection is also desirably opportunistic, performing almost as well as per-connection selection in scenarios with very little contention ($1 \leq c \leq 4$). Per-destination selection and PRNG-based methods that reserve IPIDs for non-repetition have poor performance and scalability, as their implementations rely on locking a single global resource (the hash table of destination counters, the searchable queue, or the shuffled permutation).

For space complexity, we consider all data structures that a selection method stores between IPID requests (see Table 2). Globally incrementing selection stores a single 16-bit counter. Per-connection and per-destination selection store one 16-bit counter per active connection or destination, respectively. There are ~3.7 billion unreserved IP addresses and thus there could be at most as many destination counters; the maximum number of connection counters is even larger since connections also consider source and destination ports and protocol numbers. In practice, Windows fixes a purge threshold—2^{12} destination counters in Windows 10 and 2^{15} in Windows Server—that puts a soft limit on the number of destination counters and 32-bit last access timestamps it stores before purging stale counters, though this limit can briefly be surpassed between purge sequences [44, 45]. Linux treats connection counters as acceptably small members of its socket data structures, and implements per-bucket selection by storing one 16-bit counter and one 32-bit last access timestamp for each of its 2^{11} to 2^{18} buckets. PRNG selection using a searchable queue of size $k \in [2^{12}, 2^{15}]$ stores the last k IPID values generated by the PRNG, plus any data structure used to make searching fast; for example, FreeBSD and XNU search their queues in constant time by storing an array of 2^{16} bits where the i-th bit is 1 if and only if IPID i is currently in the queue. PRNG selection using the iterated Knuth shuffle stores the current permutation of all 2^{16} IPID values plus one 16-bit index for the next value in the permutation. Finally, pure PRNG selection does not store any information between IPID requests other than the state of its PRNG.

Taken together, pure PRNG selection outperforms all but per-connection selection in terms of time and does not use any memory beyond its PRNG state; globally incrementing selection is the runner-up, outperforming the remaining methods while only using 2 bytes of memory. Per-destination selection (with an upper limit on destination counters) and the non-repeating PRNG-based methods have moderate memory footprints, but their global locks are prohibitive performance bottlenecks that cannot be removed without threatening their correctness or security guarantees. Finally, per-bucket selection achieves competitive IPID request times at the cost of potentially large (but bounded) memory usage.

5 RECOMMENDATIONS

Which IPID selection method is best? As is the case for most worthwhile questions, it depends. In this section, we synthesize insights from prior literature with the correctness, security, and performance evaluations of Section 4 into concrete recommendations for IPID selection best practices.

Many have argued for avoiding fragmentation altogether by explicitly marking packets as atomic when possible [5, 28, 34, 41, 53], e.g., when packets are small or the path MTU is discovered as part of a TCP connection. This best practice is already widely adopted. All five OSes we reviewed make packets atomic when possible, and 82.4% of packets in the CAIDA trace we used for benchmarking [12] and an estimated 99% of all network traffic is not fragmented [68].

RECOMMENDATION 1. *Any IP packet that can avoid fragmentation (e.g., with a known path MTU) should be made atomic with DF = TRUE, MF = FALSE, and Fragment Offset = 0.*

But making packets atomic does not obviate the question of how their IPIDs should be assigned. RFC 6864 [70] states that an atomic packet's IPID can be any value and should be ignored by subsequent machines. This latitude presents a previously unleveraged opportunity to secure not only the atomic packets—e.g., in the case of forced fragmentation (Section 3.2)—but also the packets that may not be able to avoid fragmentation, such as DNS and QUIC traffic sent via UDP. As we saw in Section 4, the security and performance of various IPID selection methods depend on the rate λ of IPID assignment. Choosing whether to assign IPIDs to atomic packets using a fixed value (as FreeBSD, macOS, and Linux do), a dedicated selection method (as Linux does), or the same selection method as for non-atomic packets (as OpenBSD does) changes those methods' effective IPID assignment rates, thus also changing their expected security and performance.

We propose a new approach to IPID selection based on user choice instead of OS developer hardcoding. Our analyses in Section 4 showed that although there is no universally "best" IPID selection method for every use case across all three dimensions of correctness, security, and performance, different methods achieve desirable tradeoffs for specific ranges of outgoing traffic. Thus, in this new approach, OSes would implement several IPID selection methods and expose this choice as a network setting. Users or system administrators would then estimate their rates of traffic to determine their use case and choose methods that best suit their needs, as we describe below.

RECOMMENDATION 2. *OS implementations should make IPID selection configurable (e.g., as a network setting), enabling end users and system administrators to choose selection methods for their connection-bound and non-connection-bound packets. Per-connection selection for connection-bound traffic and per-bucket selection for all other traffic can be used as the default configuration.*

Suppose, then, that an end user or system administrator can estimate their Poisson outgoing packet rate as $\lambda = \lambda_c + \lambda_n$, where λ_c is the Poisson rate of connection-bound packets and λ_n is the Poisson rate of all other packets (i.e., non-atomic and non-connection-bound atomic packets). We can use our evaluations from Section 4 to partition the space of rates (λ, λ_n) by use case and identify the best IPID selection method(s) for each. See Fig. 9 for a visual summary.

RECOMMENDATION 3. *IPID selection method(s) should be chosen based on a machine's expected rates of total outgoing traffic and non-connection-bound outgoing traffic according to Fig. 9.*

In some of the use cases we describe below, it will be advantageous to assign IPIDs to connection-bound traffic separately from the rest (Fig. 9, white hatching). In these situations, connection-bound atomic packets should be assigned IPIDs using per-connection selection, storing connection counters in the corresponding socket data structures. This is preferable to assigning them a fixed IPID value (e.g., zero) since it acts as a low-cost implementation of Postel's law—"be conservative in what
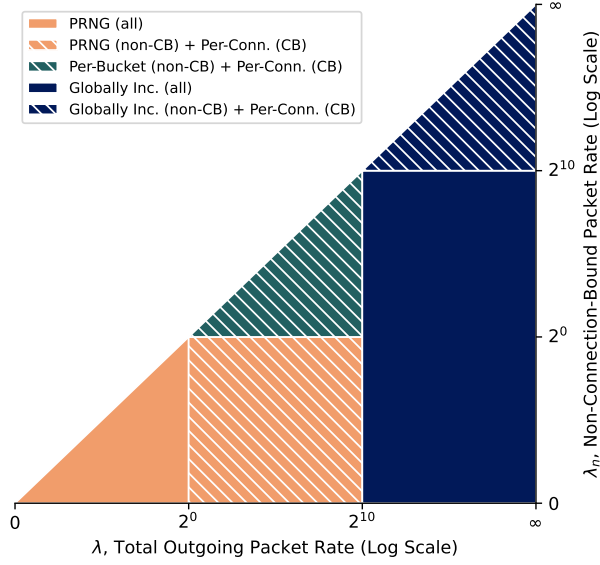
Fig. 9. *Recommended IPID Selection Method(s) by Use Case.* Depending on a machine's rates of total outgoing packets λ and non-connection-bound (non-CB) packets λ_n, globally incrementing (dark blue), per-bucket (teal), or PRNG-based selection (orange) may provide the best correctness-security-performance tradeoff. Connection-bound (CB) packets can be handled separately with per-connection selection (white hatching) or combined with the rest depending on what provides the best tradeoffs.

you send, be liberal in what you accept"—hedging against forced fragmentation by a malicious adversary or other machines fragmenting packets downstream in violation of RFC 6848.

It remains to determine, for each use, the best selection method for non-connection-bound packets and whether this method should also be used for connection-bound packets.

(1) *Slow Overall* (e.g., printers, edge devices, IoT). If the total outgoing packet rate is slow ($\lambda_n \leq \lambda \leq 2^0 \approx 1$ Mbps), our analysis shows that all methods are likely correct; per-bucket and PRNG-based selection achieve near-optimal security; and PRNG-based selection is faster than per-bucket selection as there is little to no CPU contention over IPID resources at these slow rates. Thus, PRNG-based selection can be used efficiently for all packets, regardless of whether they're connection-bound (Fig. 9, solid orange). The specific choice of PRNG-based method can be left up to user preference. Pure PRNG selection has the best performance and security but much worse correctness; between the non-repeating methods, searchable queues use less memory while iterated Knuth shuffles have shorter request times, and larger numbers of reserved IPIDs k achieve better correctness at the cost of slightly worse security.

(2) *Slow Non-Connection-Bound, Moderate Connection-Bound* (e.g., home computers, HTTPS servers handling exclusively TCP traffic). If only the non-connection-bound traffic rate is slow ($\lambda_n \leq 2^0 \approx 1$ Mbps) but the overall rate is too fast for pure PRNG selection to be correct and for non-repeating PRNG-based methods to be efficient ($\lambda > 2^0 \approx 1$ Mbps), then the connection-bound traffic should be handled separately with per-connection selection so the remaining rate is slow enough to use a non-repeating PRNG-based method efficiently (Fig. 9, hatched orange). This achieves near-optimal security for all traffic with high performance.

(3) *Moderate Overall* (e.g., home computers, general purpose servers). In the region of moderate traffic rates ($2^0 < \lambda$, $\lambda_n < 2^{10} \approx 1$ Gbps), pure PRNG selection's collision probabilities become unacceptably high and the performance bottleneck of per-destination and non-repeating PRNG-based selections' global locks cause them to scale poorly. Globally incrementing and per-bucket selection perform much better, and per-bucket's stochastic increments yield better security guarantees than global's sequential increments for these moderate rates. Those stochastic increments add the most noise for low traffic buckets, so connection-bound traffic should be handled separately with per-connection selection (Fig. 9, hatched teal).

(4) *Moderate Non-Connection-Bound, Fast Connection-Bound* (e.g., busy general purpose servers). In this use case, we reach perhaps the most surprising result of our analysis: reports of globally incrementing selection's disadvantages are greatly exaggerated. Globally incrementing selection not only scales better than all other methods under high contention (other than per-connection and pure PRNG), but also overtakes per-bucket selection in security (recall Section 4.3 for details). Thus, if the non-connection-bound packet rate is not fast enough to take advantage of globally incrementing selection alone ($\lambda_n < 2^{10} \approx 1$ Gbps) but the total outgoing packet rate is ($\lambda \geq 2^{10} \approx 1$ Gbps), then all traffic should be lumped together under globally incrementing selection (Fig. 9, solid dark blue).

(5) *Fast Non-Connection-Bound* (e.g., DNS servers). By the same logic as in the previous case, if the non-connection-bound packet rate is fast enough to benefit from globally incrementing selection on its own ($\lambda_n \geq 2^{10} \approx 1$ Gbps), then it should do so. To avoid unnecessarily burdening the single global counter and its cache coherency protocol, the connection-bound traffic can use per-connection selection (Fig. 9, hatched dark blue).

5.1 Implementation Suggestions

Altogether, we recommend that OSes implement four IPID selection methods: globally incrementing, per-connection, per-bucket, and a non-repeating PRNG-based method. (Notably, per-destination selection is never recommended, owing to its poor performance and historically poor security regardless of use case.) Against the benefits of secure and high-performance IPID selection configurable for each machine's specific use case, the costs of this proposed approach are small: some added code complexity and slight memory footprint inflation. For example, Linux already implements both per-connection and per-bucket selection; adding globally incrementing selection and a non-repeating PRNG-based method would only use 16–128 KiB more memory. FreeBSD also implements multiple methods (globally incrementing and PRNG with a searchable queue), using a user-configurable variable to choose between them.

At least in Unix-like OSes, there are standard configuration files and settings menus exposing networking variables to users. For example, Linux's variable definitions in /proc/sys/net/ipv4/ already enable runtime user configuration of various IPv4 fragmentation parameters, such as the reassembly timeout (ipfrag_time). To achieve our recommendations, users must be able to configure (1) which IPID selection method is used in general—choosing one of globally incrementing, per-bucket (default), or a non-repeating PRNG-based method—and (2) whether connection-bound traffic is handled separately by per-connection IPID selection (default: yes). OS implementation can then follow Linux's method for differentiating connection-bound and non-connection-bound traffic and FreeBSD's method for activating the user-specified IPID selection method. To aid users in choosing a suitable method for their use case, one could imagine a basic command line utility that leverages existing network monitoring tools (e.g., iftop) to estimate total and connection-bound outgoing traffic rates and then recommend a selection method according to Fig. 9.

Our final recommendation is a word of caution. In an effort to avoid creating new side channels when addressing old ones, we advise against dynamically updating IPID selection methods based on

real-time traffic rates. Recall that an off-path observer can easily infer which IPID selection method is active from observations of IPIDs over time [5, 57]. If the conditions for changing selection methods is known (as they would be for any open-source OS), any observable change in IPID patterns inherently leaks information about the status of those conditions, just as in hybrid leaks and downgrade attacks (Sections 3.1.5–3.1.6). If those conditions are further tied to something an attacker can influence (e.g., real-time traffic rates), the attacker may abuse this (e.g., by varying its probe rate) to force the target into using a selection method that it desires. Instead, if different selection methods are preferable at different times (e.g., because traffic rates vary due to diurnal patterns or weekdays vs. weekends vs. holidays), OSes could support user-defined IPID selection method schedules. This can only leak the existence of the schedule itself, but does not reveal other information about the state of the system and cannot be otherwise influenced by an off-path attacker. In short, we advocate for user-configurable IPID selection methods, not methods that can be chosen or influenced by external actors beyond the user's control.

5.2 Limitations

Our recommendations come with the following caveats and limitations. First, as discussed in detail in Section 4.1, our choice to model network traffic according to Poisson processes prioritizes unifying analytical elegance over total empirical fidelity. This is a reasonable modeling choice for large-scale aggregated traffic or short time scales, but may not capture bursty, non-stochastic patterns. In particular, our recommendation of using globally incrementing selection for scenarios with high traffic rates critically depends on continuous, rapid increments of the global counter; periods of slow traffic remain vulnerable to idle scans and other exploits that leverage easily predictable IPIDs. A critical evaluation of our analytical results using simulated or deployed networking environments would be a valuable direction for future work.

6 CONCLUSION

In this survey, we collected a complete history of IPID-based exploits and the corresponding OS changes to IPID selection, categorizing these off-path measurements and attacks as either *probe comparisons* aiming to infer information about other machines through changes in IPIDs over time or *fragment injections* aiming to replace legitimate fragments with malicious ones during reassembly, causing packets to be poisoned or discarded. We then presented the first comparative analysis of all seven IPID selection methods, formally analyzing their relative correctness and security and empirically evaluating their performance. Of these evaluations, the most surprising is that globally incrementing selection—the first, simplest, and most dismissed of all IPID selection methods—is in fact the most collision-avoidant, secure, and performant choice for non-connection-bound packets when the rates of outgoing packets are very high. Finally, we proposed a new approach to IPID selection that shifts the focus from developers making the "best" choice for their OS to users making the best choice for their use case across different ranges of total outgoing packet rates and non-connection-bound outgoing packet rates (see Fig. 9). It is our hope that further comparative analyses and theoretical evaluations of basic networking protocols will reveal similar best practices that can strengthen the correctness, security, and performance of all Internet-connected devices.

SOFTWARE ARTIFACTS

Source code for the comparative analysis and performance benchmark in Section 4 is available at https://github.com/DaymudeLab/IPIDSurvey-Code.

ACKNOWLEDGMENTS

REFERENCES

[1] Geoffrey Alexander, Antonio M. Espinoza, and Jedidiah R. Crandall. 2019. Detecting TCP/IP Connections via IPID Hash Collisions. *Proceedings on Privacy Enhancing Technologies* 2019, 4 (2019), 311–328. https://doi.org/10.2478/popets-2019-0071

[2] antirez. 1998. New TCP Scan Method. Bugtraq Mailing List. https://seclists.org/bugtraq/1998/Dec/79.

[3] Ivan Arce and Emiliano Kargieman. 1997. *BIND Vulnerabilities and Solutions*. Security Advisory. Secure Networks Inc. and CORE Seguridad de la Informacion. https://marc.info/?l=best-of-security&m=96843707620680.

[4] Soshant Bali and Victor S. Frost. 2007. An Algorithm for Fitting MMPP to IP Traffic Traces. *IEEE Communications Letters* 11, 2 (2007), 207–209. https://doi.org/10.1109/LCOMM.2007.061619

[5] Steven M. Bellovin. 1989. Security Problems in the TCP/IP Protocol Suite. *ACM SIGCOMM Computer Communication Review* 19, 2 (1989), 32–48. https://doi.org/10.1145/378444.378449

[6] Steven M. Bellovin. 2002. A Technique for Counting NATted Hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*. ACM, 267–272. https://doi.org/10.1145/637201.637243

[7] Robert Beverly, William Brinkmeyer, Matthew Luckie, and Justin P. Rohrer. 2013. IPv6 Alias Resolution via Induced Fragmentation. In *Passive and Active Measurement (Lecture Notes in Computer Science, Vol. 7799)*. Springer, 155–165. https://doi.org/10.1007/978-3-642-36516-4_16

[8] Robert Beverly, Matthew Luckie, Lorenza Mosley, and K.C. Claffy. 2015. Measuring and Characterizing IPv6 Router Availability. In *Passive and Active Measurement (Lecture Notes in Computer Science, Vol. 8995)*. Springer, 123–135. https://doi.org/10.1007/978-3-319-15509-8_10

[9] Vasanthakumar N. Bhat. 1994. Renewal Approximations of the Switched Poisson Processes and Their Applications to Queueing Systems. *The Journal of the Operational Research Society* 45, 3 (1994), 345–353. https://doi.org/10.2307/2584168

[10] R. Braden. 1989. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. RFC Editor. https://doi.org/10.17487/RFC1122

[11] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. 2018. Domain Validation++ For MitM-Resilient PKI. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2060–2076. https://doi.org/10.1145/3243734.3243790

[12] CAIDA. 2019. The CAIDA UCSD Anonymized Internet Traces - 2019. https://www.caida.org/catalog/datasets/passive_dataset/.

[13] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. 2003. Internet Traffic Tends Toward Poisson and Independent as the Load Increases. In *Nonlinear Estimation and Classification*, David D. Denison, Mark H. Hansen, Christopher C. Holmes, Bani Mallick, and Bin Yu (Eds.). Lecture Notes in Statistics, Vol. 171. Springer, New York, NY, USA, 83–109. https://doi.org/10.1007/978-0-387-21579-2_6

[14] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. 2016. Off-Path TCP Exploits: Global Rate Limit Considered Dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 209–225.

[15] Neal Cardwell, Stefan Savage, and Thomas Anderson. 2000. Modeling TCP Latency. In *IEEE INFOCOM 2000 - IEEE Conference on Computer Communications*, Vol. 3. IEEE, 1742–1751. https://doi.org/10.1109/INFCOM.2000.832574

[16] Bor-Sen Chen, Sen-Chueh Peng, and Ku-Chen Wang. 2000. Traffic Modeling, Prediction, and Congestion Control for High-Speed Networks: A Fuzzy AR Approach. *IEEE Transactions on Fuzzy Systems* 8, 5 (2000), 491–508. https://doi.org/10.1109/91.873574

[17] Weifeng Chen, Yong Huang, Bruno F. Ribeiro, Kyoungwon Suh, Honggang Zhang, Edmundo de Souza e Silva, Jim Kurose, and Don Towsley. 2005. Exploiting the IPID Field to Infer Network Path and End-System Characteristics. In *Passive and Active Network Measurement (Lecture Notes in Computer Science, Vol. 3431)*. Springer, 108–120. https://doi.org/10.1007/978-3-540-31966-5_9

[18] Marco de Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O. de Vivo. 1999. A Review of Port Scanning Techniques. *ACM SIGCOMM Computer Communication Review* 29, 2 (1999), 41–48. https://doi.org/10.1145/505733.505737

[19] S. Deering and R. Hinden. 2017. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. RFC Editor. https://doi.org/10.17487/RFC8200

[20] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. 2009. Cryptanalysis of the Random Number Generator of the Windows Operating System. *ACM Transactions on Information and System Security* 13, 1 (2009), 1–32. https://doi.org/10.1145/1609956.1609966

[21] Roya Ensafi. 2014. *Advanced Network Inference Techniques Based on Network Protocol Stack Information Leaks*. Ph. D. Dissertation. University of New Mexico, Albequerque, NM, USA.

[22] Roya Ensafi, Jeffrey Knockel, Geoffrey Alexander, and Jedidiah R. Crandall. 2014. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *Passive and Active Measurement (Lecture Notes in Computer Science, Vol. 8362)*. Springer, 109–118. https://doi.org/10.1007/978-3-319-04918-2_11

[23] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. 2010. Idle Port Scanning and Non-Interference Analysis of Network Protocol Stacks Using Model Checking. In *19th USENIX Security Symposium (USENIX Security 10)*. USENIX Association, 1–16.

[24] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R. Crandall. 2015. Analyzing the Great Firewall of China Over Space and Time. In *Proceedings on Privacy Enhancing Technologies*, Vol. 2015. 61–76. https://doi.org/10.1515/popets-2015-0005

[25] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. 2020. Off-Path TCP Exploits of the Mixed IPID Assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1323–1335. https://doi.org/10.1145/3372297.3417884

[26] Xuewei Feng, Qi Li, Kun Sun, Chuanpu Fu, and Ke Xu. 2022. Off-Path TCP Hijacking Attacks via the Side Channel of Downgraded IPID. *IEEE/ACM Transactions on Networking* 30, 1 (2022), 409–422. https://doi.org/10.1109/TNET.2021.3115517

[27] Xuewei Feng, Qi Li, Kun Sun, Ke Xu, Baojun Liu, Xiaofeng Zheng, Qiushi Yang, Haixin Duan, and Zhiyun Qian. 2022. PMTUD Is Not Panacea: Revisiting IP Fragmentation Attacks against TCP. In *Proceedings of the 2022 Network and Distributed System Security Symposium*. Internet Society, 1–18. https://doi.org/10.14722/ndss.2022.24381

[28] Yossi Gilad and Amir Herzberg. 2011. Fragmentation Considered Vulnerable: Blindly Intercepting and Discarding Fragments. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*. USENIX Association, 1–10.

[29] Yossi Gilad and Amir Herzberg. 2012. Off-Path Attacking the Web. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. USENIX Association, 1–12.

[30] Yossi Gilad and Amir Herzberg. 2013. Fragmentation Considered Vulnerable. *ACM Transactions on Information and System Security* 15, 4 (2013), 16:1–16:31. https://doi.org/10.1145/2445566.2445568

[31] Yossi Gilad and Amir Herzberg. 2014. Off-Path TCP Injection Attacks. *ACM Transactions on Information and System Security* 16, 4 (2014), 1–32. https://doi.org/10.1145/2597173

[32] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. 2006. Analysis of the Linux Random Number Generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 1–15. https://doi.org/10.1109/SP.2006.5

[33] Mark Handley, Vern Paxson, and Christian Kreibich. 2001. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *10th USENIX Security Symposium (USENIX Security 01)*. USENIX Association, 1–17.

[34] Amir Herzberg and Haya Shulman. 2013. Fragmentation Considered Poisonous, or: One-Domain-to-Rule-Them-All.org. In *2013 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 224–232. https://doi.org/10.1109/CNS.2013.6682711

[35] Amir Herzberg and Haya Shulman. 2013. Vulnerable Delegation of DNS Resolution. In *Computer Security – ESORICS 2013 (Lecture Notes in Computer Science, Vol. 8134)*. Springer, 219–236. https://doi.org/10.1007/978-3-642-40203-6_13

[36] J. R. M. Hosking. 1981. Fractional Differencing. *Biometrika* 68, 1 (1981), 165–176. https://doi.org/10.1093/biomet/68.1.165

[37] Fengyuan Huang, Yifan Yang, Zhenzhong Yang, Bingnan Hou, Yingwen Chen, and Zhiping Cai. 2025. A Closer Look at IPv6 IP-ID Behavior in the Wild. In *Passive and Active Measurement (Lecture Notes in Computer Science, Vol. 15567)*. Springer, 30–43. https://doi.org/10.1007/978-3-031-85960-1_2

[38] Douglas M. Jennewein, Johnathan Lee, Chris Kurtz, William Dizon, Ian Shaeffer, Alan Chapman, Alejandro Chiquete, Josh Burks, Amber Carlson, Natalie Mason, Arhat Kobawala, Thirugnanam Jagadeesan, Praful Bhargav Basani, Torey Battelle, Rebecca Belshe, Deb McCaffrey, Marisa Brazil, Chaitanya Inumella, Kirby Kuznia, Jade Buzinski, Dhruvil Deepakbhai Shah, Sean M. Dudley, Gil Speyer, and Jason Yalim. 2023. The Sol Supercomputer at Arizona State University. In *Practice and Experience in Advanced Research Computing*. ACM, Portland, OR, USA, 296–301. https://doi.org/10.1145/3569951.3597573

[39] Thomas Karagiannis, Mart Molle, and Michalis Faloutsos. 2004. Long-Range Dependence: Ten Years of Internet Traffic Modeling. *IEEE Internet Computing* 8, 5 (2004), 57–64. https://doi.org/10.1109/MIC.2004.46

[40] Thomas Karagiannis, Mart Molle, Michalis Faloutsos, and Andre Broido. 2004. A Nonstationary Poisson View of Internet Traffic. In *IEEE INFOCOM 2004 - IEEE Conference on Computer Communications*, Vol. 3. IEEE, 1558–1569.

https://doi.org/10.1109/INFCOM.2004.1354569

[41] Christopher A. Kent and Jeffrey C. Mogul. 1987. Fragmentation Considered Harmful. *SIGCOMM Computer Communication Review* 17, 5 (1987), 390–401. https://doi.org/10.1145/55483.55524

[42] Amit Klein. 2007. *OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability.* Technical Report. Trusteer. 1–48 pages.

[43] Amit Klein. 2021. Cross Layer Attacks and How to Use Them (for DNS Cache Poisoning, Device Tracking and More). In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1179–1196. https://doi.org/10.1109/SP40001.2021.00054

[44] Amit Klein. 2022. Subverting Stateful Firewalls with Protocol States. In *Proceedings of the 2022 Network and Distributed System Security Symposium*. Internet Society, 1–18. https://doi.org/10.14722/ndss.2022.23037

[45] Amit Klein. 2024. Private Communication.

[46] Amit Klein and Benny Pinkas. 2019. From IP ID to Device ID and KASLR Bypass. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1063–1080.

[47] Jeffrey Knockel and Jedidiah R. Crandall. 2014. Counting Packets Sent Between Arbitrary Internet Hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*. USENIX Association, 1–8.

[48] Donald E. Knuth. 1969. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms.* Vol. 2. Addison-Wesley, Reading, MA, USA.

[49] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (1973), 613–615. https://doi.org/10.1145/362375.362389

[50] Will E. Leland, Walter Willinger, Murad S. Taqqu, and Daniel V. Wilson. 1995. On the Self-Similar Nature of Ethernet Traffic. *ACM SIGCOMM Computer Communication Review* 25, 1 (1995), 202–213. https://doi.org/10.1145/205447.205464

[51] lkm. 2007. Remote Blind TCP/IP Spoofing. *Phrack Magazine* 64 (2007), 13. http://phrack.org/issues/64/13.html#article.

[52] Gordon "Fyodor" Lyon. 2009. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* Nmap Project, Sunnyvale, CA.

[53] Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, and Sharon Goldberg. 2016. Attacking the Network Time Protocol. In *Proceedings of the 2016 Network and Distributed System Security Symposium*. Internet Society, 1–15. https://doi.org/10.14722/ndss.2016.23090

[54] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Raliton, Ron Deibert, and Vern Paxson. 2015. An Analysis of China's "Great Cannon". In *5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 15)*. USENIX Association, 1–11.

[55] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review* 27, 3 (1997), 67–82. https://doi.org/10.1145/263932.264023

[56] João Paulo S. Medeiros, Agostinho M. Brito, and Paulo S. Motta Pires. 2010. An Effective TCP/IP Fingerprinting Technique Based on Strange Attractors Classification. In *Data Privacy Management and Autonomous Spontaneous Security (Lecture Notes in Computer Science, Vol. 5939)*. Springer, 208–221. https://doi.org/10.1007/978-3-642-11207-2_16

[57] Sophon Mongkolluksamee, Kensuke Fukuda, and Panita Pongpaibool. 2012. Counting NATted Hosts by Observing TCP/IP Field Behaviors. In *2012 IEEE International Conference on Communications (ICC)*. IEEE, 1265–1270. https://doi.org/10.1109/ICC.2012.6364596

[58] Mathias Morbitzer. 2013. *TCP Idle Scans in IPv6.* Master's thesis. Radboud University Nijmegen, Nijmegen, Netherlands.

[59] Liran Orevi, Amir Herzberg, and Haim Zlatokrilov. 2018. DNS-DNS: DNS-Based De-NAT Scheme. In *Cryptology and Network Security (Lecture Notes in Computer Science, Vol. 11124)*. Springer, 69–88. https://doi.org/10.1007/978-3-030-00434-7_4

[60] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. 1998. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. *ACM SIGCOMM Computer Communication Review* 28, 4 (1998), 303–314. https://doi.org/10.1145/285243.285291

[61] Travis Palmer and Brian Somers. 2019. "First-Try" DNS Cache Poisoning with IPv4 and IPv6 Fragmentation. DEF CON 27. https://media.defcon.org/DEF%20CON%2027/DEF%20CON%2027%20presentations/DEFCON-27-Travis-Palmer-First-try-DNS-Cache-Poisoning-with-IPv4-and-IPv6-Fragmentation.pdf.

[62] Vern Paxson and Sally Floyd. 1995. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking* 3, 3 (1995), 226–244. https://doi.org/10.1109/90.392383

[63] Paul Pearce, Roya Ensafi, Frank Li, Nick Feamster, and Vern Paxson. 2017. Augur: Internet-Wide Detection of Connectivity Disruptions. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 427–443. https://doi.org/10.1109/SP.2017.55

[64] Jon Postel. 1981. *Internet Protocol.* RFC 791. RFC Editor. https://doi.org/10.17487/RFC0791

[65] Zhiyun Qian and Z. Morley Mao. 2012. Off-Path TCP Sequence Number Inference Attack - How Firewall Middleboxes Reduce Security. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 347–361. https://doi.org/10.1109/SP.2012.29

[66] Zhiyun Qian, Z. Morley Mao, Yinglian Xie, and Fang Yu. 2010. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *2010 IEEE Symposium on Security and Privacy.* IEEE, 207–222. https://doi.org/10.1109/SP.2010.42

[67] Christoph Schuba. 1993. *Addressing Weaknesses in the Domain Name System Protocol.* Master's thesis. Purdue University, West Lafayette, IN, USA.

[68] Colleen Shannon, David Moore, and K. C. Claffy. 2002. Beyond Folklore: Observations on Fragmented Traffic. *IEEE/ACM Transactions on Networking* 10, 6 (2002), 709–720. https://doi.org/10.1109/TNET.2002.805028

[69] P. Srisuresh and M. Holdrege. 1999. *IP Network Address Translator (NAT) Terminology and Considerations.* RFC 2663. RFC Editor. https://doi.org/10.17487/rfc2663

[70] J. Touch. 2013. *Updated Specification of the IPv4 ID Field.* RFC 6864. RFC Editor. https://doi.org/10.17487/RFC6864

[71] Xu Zhang, Jeffrey Knockel, and Jedidiah R. Crandall. 2018. ONIS: Inferring TCP/IP-Based Trust Relationships Completely Off-Path. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications.* IEEE, 2069–2077. https://doi.org/10.1109/INFOCOM.2018.8486426

[72] Hong Zhao and Yun-Qing Shi. 2013. Detecting Covert Channels in Computer Networks Based on Chaos Theory. *IEEE Transactions on Information Forensics and Security* 8, 2 (2013), 273–282. https://doi.org/10.1109/TIFS.2012.2231861

[73] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. 2020. Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 577–593.

A ANALYTICAL DERIVATIONS

A.1 Correctness Derivations

To capture the stochastic nature of network traffic, we assume a server S sends packets according to a Poisson process with rate $\lambda > 0$. We further assume a worst-case scenario in which S sends all its packets to the same client C using the same protocol, every packet is fragmented, and the reassembly buffer of C has unlimited size; thus, every packet has the potential to cause a collision. Let N be the corresponding random variable counting the number of packets "simultaneously in transit" from S to C—i.e., those sent by S whose fragments have not yet been reassembled or evicted by C—and let unit time be calibrated such that $\mathrm{E}[N] = \lambda$. Then N is Poisson-distributed, characterized by probability mass, cumulative distribution, and survival functions:

$$\mathrm{pmf}(N, \lambda) = \frac{\lambda^N e^{-\lambda}}{N!} \tag{4}$$

$$\mathrm{cdf}(N, \lambda) = \sum_{n=0}^{N} \mathrm{pmf}(n, \lambda) \tag{5}$$

$$\mathrm{sf}(N, \lambda) = \sum_{n=N+1}^{\infty} \mathrm{pmf}(n, \lambda) = 1 - \mathrm{cdf}(N, \lambda) \tag{6}$$

A.1.1 Globally Incrementing. Globally incrementing IPIDs increment sequentially, so the only way for a collision to occur is if all possible IPID values are exhausted and at least one is reused. Thus, the probability of S producing a collision at C among $N = n$ packets simultaneously in transit is

$$\Pr[\text{collision} \mid N = n] = \begin{cases} 0 & \text{if } n \leq 2^{16}; \\ 1 & \text{otherwise.} \end{cases} \tag{7}$$

Substituting Eq. 7 into Eq. 1 yields

$$\Pr[\text{collision}] = \sum_{n=2^{16}+1}^{\infty} \mathrm{pmf}(n, \lambda) = \mathrm{sf}(2^{16}, \lambda). \tag{8}$$

A.1.2 Per-Connection and Per-Destination. Per-connection and per-destination counters also increment sequentially and thus must exhaust all IPID values before a collision can occur. Thus,

Eq. 8 also describes the probability of S producing a collision at C when using per-connection or per-destination selection.

A.1.3   Per-Bucket (Linux). All packets sent from S to C via the same protocol are hashed to the same bucket, so the total number of buckets has no bearing on the probability of collision. However, bucket counters are not sequentially incrementing like those in the above methods. Instead, Linux's stochastic increments make it possible to skip IPID values, cycle back around, and collide in fewer than 2^{16} packets. Let Δ be a random variable representing the number of system ticks since S last sent a packet; thus, the next bucket increment is chosen uniformly at random from $\{1, \ldots, \Delta\}$. Since packet transmissions occur according to a Poisson process with rate λ, Δ is exponentially distributed with mean t/λ, where $t > 0$ is the number of system ticks per unit time. For this analysis, we set $t = 3$, assuming that packets spend ~10 ms in transit and there are ~300 ticks/s.[4]

Unfortunately, even with these carefully constructed assumptions, $\Pr[\text{collision} \mid N = n]$ is challenging to bound analytically—let alone to calculate explicitly—because each event is dependent on all others. Specifically, whether a new IPID produces a collision depends on the stochastic increments that produced all preceding IPIDs. Thus, for the sake of comparison with other methods, we estimate these conditional probabilities by simulating the generation of IPIDs via stochastic increments and then computing the overall probability of collision using Eq. 1.[5]

A.1.4   PRNG (searchable queue of size k). For the purposes of analysis, we assume that the PRNG algorithm in use sufficiently approximates a uniform distribution over all possible IPID values; i.e., any given IPID is chosen with probability $1/2^{16}$. In the setting where no recent IPIDs are stored (i.e., when $k = 0$) and the number of packets simultaneously in transit is $N = n$, the probability of S producing a collision at C reduces to the birthday problem with n "people" (the packets) and 2^{16} "days" (the IPIDs).[6] More generally, with a searchable queue of size $k \geq 0$, we are guaranteed that the next IPID will be distinct from the last k generated IPIDs. If $n > k$, the remaining $n - k$ IPIDs may collide with any newly generated IPID according to the birthday problem, yielding

$$\Pr[\text{collision} \mid N = n] = \begin{cases} 0 & \text{if } n \leq k; \\ 1 - \prod_{i=0}^{n-k-1}\left(1 - \frac{i}{2^{16}-k}\right) & \text{if } k < n \leq 2^{16}; \\ 1 & \text{if } n > 2^{16}. \end{cases} \quad (9)$$

Substituting Eq. 9 into Eq. 1 yields

$$\Pr[\text{collision}] = \sum_{n=k+1}^{2^{16}}\left(1 - \prod_{i=0}^{n-k-1}\left(1 - \frac{i}{2^{16}-k}\right)\right) \cdot \text{pmf}(n, \lambda) + \sum_{n=2^{16}+1}^{\infty} \text{pmf}(n, \lambda)$$

$$= \sum_{n=k+1}^{2^{16}}\left(1 - \prod_{i=0}^{n-k-1}\left(1 - \frac{i}{2^{16}-k}\right)\right) \cdot \text{pmf}(n, \lambda) + \text{sf}(2^{16}, \lambda). \quad (10)$$

---

[4]The assumption of 300 system ticks/s is consistent with Arch Linux's implementation. Ubuntu is slightly slower, at 250 ticks/s. We chose Arch's larger value because it generates larger stochastic increments and thus higher probabilities of collision, suitable for a worst-case analysis.

[5]A careful reader might observe that the infinite sum in Eq. 1 cannot be computed directly. Our simulations instead use the finite interval of $n \subset [0, \infty)$ containing all non-negligible Poisson probability mass; i.e., all n such that $\text{pmf}(n, \lambda) > 5 \times 10^{-324}$. We reuse this trick when evaluating Eqs. 14 and 17.

[6]The OpenBSD, FreeBSD, and macOS implementations of PRNG selection reserve zero as a special IPID that is never returned (Section 2.3). This detail is omitted for clarity, but is easily addressed by replacing 2^{16} total IPIDs with $2^{16} - 1$ non-zero IPIDs.

A.1.5 PRNG (iterated Knuth shuffle reserving k IPIDs). Supposing the permutation of all 2^{16} IPIDs is initialized uniformly at random and any returned IPID is swapped into a position chosen uniformly at random from among the $2^{16} - k$ previous positions in the permutation (including its own), the probability that any given IPID x will be returned next is

$$\Pr\left[\text{next IPID} = x\right] = \begin{cases} 0 & \text{if } x \in \text{last } k \text{ IPIDs;} \\ \frac{1}{2^{16}-k} & \text{otherwise.} \end{cases} \tag{11}$$

This implies that the next IPID appears uniform at random over all IPIDs except the last k returned. So, just as for a searchable queue of size k, no IPID will collide with any of the last k returned IPIDs and, if $n > k$, the remaining $n - k$ IPIDs may collide with any newly returned IPID according to the birthday problem. Thus, Eq. 10 also describes the collision probability for this method.

A.1.6 PRNG (pure, no reserved IPIDs). It is easily seen that pure PRNG selection—generating IPIDs uniformly at random over all 2^{16} possible values—is a special case of the previous two methods when $k = 0$. Simplifying Eq. 10 for this case yields

$$\Pr\left[\text{collision}\right] = \sum_{n=1}^{2^{16}} \left(1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{2^{16}}\right)\right) \cdot \text{pmf}(n, \lambda) + \text{sf}(2^{16}, \lambda). \tag{12}$$

A.2 Security Derivations

Recall from Section 4.3 that we model an off-path adversary A trying to predict the next IPID generated by a particular IPID resource $i \in \{1, \ldots, r\}$ (i.e., a counter or PRNG) on a server S. Server S sends packets according to a Poisson process with rate $\lambda > 0$ and its IPID resources i assign IPIDs with rates $\lambda \geq \lambda_i \geq 0$ where $\sum_{i=1}^{r} \lambda_i = \lambda$. Here, we analyze the probability of an adversarial guess against a particular resource i as a function of λ_i. Let N_i be the Poisson-distributed random variable counting the number of packets S sent in the last unit time whose IPIDs were assigned by resource i, and let time be calibrated such that $\text{E}\left[N_i\right] = \lambda_i$.

A.2.1 Globally Incrementing. Globally incrementing selection uses one global counter for all IPID generation, so $r = 1$ and $\lambda_i = \lambda$. The adversary A can check the status of this counter by probing S and examining the resulting IPID. This tells A the status of the global counter one unit time ago, when the reply was sent; w.l.o.g., suppose the IPID of this reply was 0. Since the global counter increments once per packet sent by S and there have been $N_i = N$ packets sent by S in the last unit time, the current status of the global counter is $N \bmod 2^{16}$ and the next IPID will be $N + 1 \bmod 2^{16}$. Thus, the probability that the next IPID is x is

$$\Pr\left[\text{next IPID} = x\right] = \Pr\left[N + 1 \equiv x \bmod 2^{16}\right]$$

$$= \sum_{n=0}^{\infty} \Pr\left[N + 1 \equiv x \bmod 2^{16} \mid N = n\right] \cdot \Pr\left[N = n\right]$$

$$= \sum_{n=0}^{\infty} \Pr\left[n + 1 \equiv x \bmod 2^{16}\right] \cdot \text{pmf}(n, \lambda), \tag{13}$$

where $\Pr\left[N = n\right] = \text{pmf}(n, \lambda)$ follows from the fact that N is Poisson-distributed. Substituting Eq. 13 into Eq. 2 yields

$$\Pr\left[\text{adv. guess}\right] = \max_{G \subset [2^{16}] \,:\, |G|=g} \left\{\sum_{x \in G} \sum_{n=0}^{\infty} \Pr\left[n + 1 \equiv x \bmod 2^{16}\right] \cdot \text{pmf}(n, \lambda)\right\}. \tag{14}$$
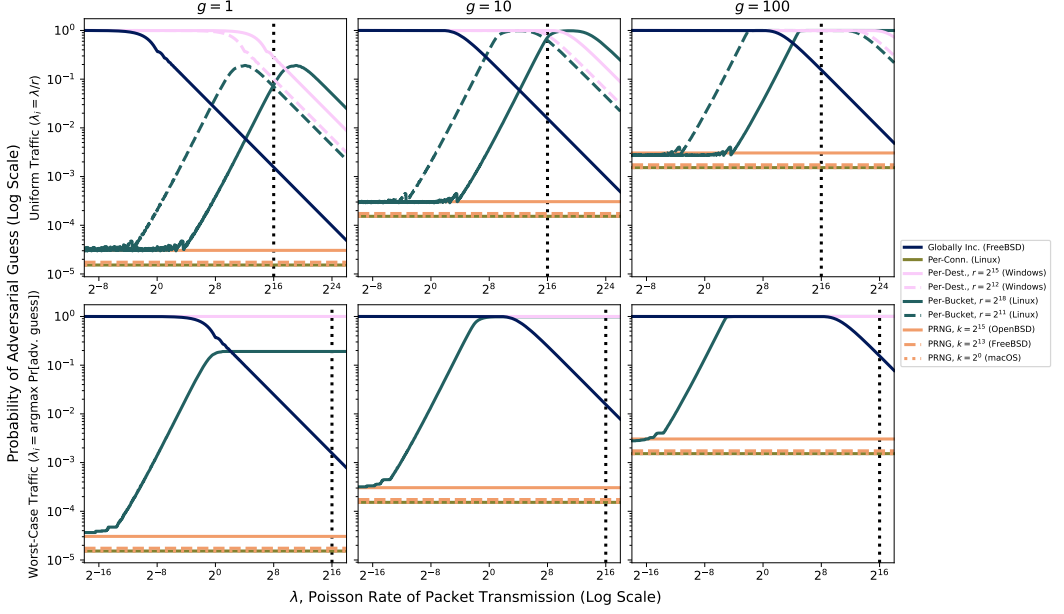
Fig. 10. *Comparison of IPID Selection Methods' Security with Multiple Adversarial Guesses.* Adversarial guess probabilities for $g \in \{1, 10, 100\}$ guesses for globally incrementing (dark blue, Eq. 14), per-connection (olive, Eq. 15), per-destination (pink, also Eq. 14), per-bucket (teal, Eq. 17 with simulation), and PRNG-based (orange, Eq. 19) IPID selection as a function of λ, the Poisson rate of packet transmission. For selection methods that use $r \geq 1$ counters, we compare the case where traffic is uniformly distributed to each counter i with rate $\lambda_i = \lambda/r$ (top) and the worst-case scenario when traffic is allocated to λ_i such that the adversarial guess probability is maximized (bottom). The total number of distinct IPIDs, 2^{16}, is shown as a black dotted line.

A.2.2   Per-Connection. Per-connection selection defines one randomly initialized counter i per connection, which is specific to the IP addresses and ports of S and its client C. There are no known methods for the adversary $A \notin \{S, C\}$ to obtain any information about the connection counter S uses to assign IPIDs to packets sent to C. Thus, regardless of the rate λ_i, adversary A can do no better than make g distinct guesses uniformly at random over all 2^{16} possible IPIDs:

$$\Pr[\text{adv. guess}] = \max_{G \subset [2^{16}] \, : \, |G|=g} \left\{ \sum_{x \in G} \Pr[\text{next IPID} = x] \right\} = \max_{G \subset [2^{16}] \, : \, |G|=g} \left\{ \sum_{x \in G} \frac{1}{2^{16}} \right\} = \frac{g}{2^{16}}. \quad (15)$$

Recall from Section 3.1.6 that downgrade attacks can force S to use some selection method other than per-connection if one exists [25–27]. In this case, the probability of an adversarial guess reduces to that of the downgraded selection method. Alternatively, if some future method enables A to probe connection counters, the probability of an adversarial guess reduces to that of globally incrementing selection (Eq. 14), but with λ replaced by λ_i.

A.2.3   Per-Destination. Per-destination selection uses one counter i per destination IP address and protocol number. As we reviewed in Sections 3.1.4, 3.2.1, and 3.2.2, there have been many successful methods for probing destination counters [28, 30, 34, 35, 47], each specific to a particular OS implementation that was patched after disclosure. For the purposes of this analysis, we assume the adversary can probe destination counters, focusing on the fundamental security of per-destination selection over the difficulty of accessing its counters. The adversary can then use

this information just as it did for globally incrementing selection to make inferences about the sequentially incrementing counter, so Eq. 14—after replacing λ with λ_i—also describes the probability of an adversarial guess for per-destination selection. In Figs. 7 and 10, we plot per-destination's adversarial guess probabilities for $r = 2^{12}$ and $r = 2^{15}$ active destinations since these are the purge thresholds for Windows 10 and Windows Server, respectively [45].

A.2.4 Per-Bucket (Linux). Per-bucket selection uses one counter per bucket. We assume the adversary A can probe any bucket counter i on server S that it desires by constructing a packet that hashes to that particular bucket.[7] W.l.o.g., suppose the IPID of this reply is 0. The bucket counter's current value depends on the number and timing of packets that hashed into this bucket in the unit time that elapsed since S replied to A.

Recall that, for this method, $N_i \geq 0$ is the number of packets sent by S with IPIDs assigned by bucket counter i per unit time. For each $j \in \{1, \ldots, N_i\}$, let c_j be the increment value of bucket counter i resulting in the IPID of the j-th packet sent after the reply to A. Thus, the probability that the next IPID is x is

$$
\begin{aligned}
\Pr\left[\text{next IPID} = x\right] &= \Pr\left[\sum_{j=1}^{N_i+1} c_j \equiv x \bmod 2^{16}\right] \\
&= \sum_{n=0}^{\infty} \Pr\left[\sum_{j=1}^{N_i+1} c_j \equiv x \bmod 2^{16} \mid N_i = n\right] \cdot \Pr\left[N = n\right] \\
&= \sum_{n=0}^{\infty} \Pr\left[\sum_{j=1}^{n+1} c_j \equiv x \bmod 2^{16}\right] \cdot \text{pmf}(n, \lambda_i),
\end{aligned}
\tag{16}
$$

where $\Pr\left[N_i = n\right] = \text{pmf}(n, \lambda_i)$ follows from the fact that N_i is Poisson-distributed. Substituting Eq. 16 into Eq. 2 yields

$$
\Pr\left[\text{adv. guess}\right] = \max_{G \subset [2^{16}] \,:\, |G|=g} \left\{ \sum_{x \in G} \sum_{n=0}^{\infty} \Pr\left[\sum_{j=1}^{n+1} c_j \equiv x \bmod 2^{16}\right] \cdot \text{pmf}(n, \lambda_i) \right\}.
\tag{17}
$$

Unlike globally incrementing selection where each $c_j = 1$, each per-bucket increment c_j is chosen uniformly at random from a range $\{1, \ldots, \Delta_j\}$, where—as for our analysis of per-bucket collision probabilities in Section A.1.3—Δ_j is an exponentially distributed random variable with mean t/λ_i, where $t = 3$ is the number of system ticks per unit time. This again makes formal analysis difficult; we thus use simulations to estimate $\Pr\left[\sum_{j=1}^{n+1} c_j \equiv x \bmod 2^{16}\right]$ via sampling and then compute the overall probability of an adversarial guess using Eq. 17. In Figs. 7 and 10, we plot these adversarial guess probabilities for $r = 2^{11}$ and $r = 2^{18}$ buckets since these are Linux's minimum and maximum number of buckets, respectively.

A.2.5 PRNG (searchable queue of size k). We again assume (as in our correctness analysis in Section A.1.4) that the PRNG algorithm in use sufficiently approximates a uniform distribution over all 2^{16} IPIDs. We additionally assume that, in the worst case, the adversary knows the IPIDs of

---

[7]Supposing that the bucket hash function appears to assign packets to buckets uniformly at random, the probability that at least one of an adversary's a IPv4 addresses will hash into a target bucket among r total buckets is $1 - (1 - 1/r)^a$. Even after Linux increased its maximum number of buckets to $r = 262{,}144$ (i.e., 2^{18}) [44], an adversary controlling $a = 10{,}000$ IPv4 addresses has a 3.74% chance of finding a hash collision, $a = 100{,}000$ yields a 31.7% chance, and $a = 1{,}000{,}000$ yields a 97.8% chance. These counts are similar to the sizes of large observed botnets. Machines with less RAM have fewer buckets, yielding higher hash collision probabilities at lower IP address counts.

the last k packets S sent; i.e., it knows which IPIDs are in the searchable queue. It is easy to see that, regardless of the rate λ, the probability that the next IPID is x is

$$\Pr\left[\text{next IPID} = x\right] = \begin{cases} 0 & \text{if } x \in \text{queue}; \\ \frac{1}{2^{16}-k} & \text{otherwise}. \end{cases} \tag{18}$$

Substituting Eq. 18 into Eq. 2, we find that the adversary can do no better than make g distinct guesses uniformly at random over the $2^{16} - k$ non-queued IPIDs:

$$\Pr\left[\text{adv. guess}\right] = \max_{G \subset [2^{16}] \,:\, |G|=g} \left\{ \sum_{x \in G} \Pr\left[\text{next IPID} = x\right] \right\} = \min\left\{ \frac{g}{2^{16}-k}, 1 \right\}. \tag{19}$$

A.2.6 PRNG (iterated Knuth shuffle reserving k IPIDs). We again assume the worst-case scenario that the adversary knows the IPIDs of the last k packets that S sent; i.e., it knows which IPIDs cannot be returned next. In Section A.1.5, we showed that the next IPID generated by an iterated Knuth shuffle is equally likely to be any IPID except the last k returned (Eq. 11), implying that the adversary can do no better than make g distinct guesses uniformly at random over the $2^{16} - k$ unreserved IPIDs. Thus, Eq. 19 also describes the probability of adversarial guess for this method.

A.2.7 PRNG (pure, no reserved IPIDs). As in our correctness analysis (Section A.1.6), it is easily seen that pure PRNG selection is a special case of the other PRNG-based methods when $k = 0$. Since every IPID is chosen uniformly at random from among all 2^{16} possible values, the adversary can do no better than guess at random. Thus, Eq. 19 (with $k = 0$) also describes the probability of adversarial guess for this method.

B BENCHMARK DETAILS

B.1 Implementation Details

Algorithm 1 details our implementations of the various IPID selection methods for the benchmarking results reported in Section 4.4. We give some context for our design decisions below.

Globally Incrementing. We assume the single global counter shared by all CPUs is atomic, meaning it can be accessed and incremented by any one CPU without explicit locking. Specifically, we use C++'s `std::atomic<uint16_t>` which implements the necessary cache coherency protocols.

Per-Connection. In practice, Linux handles TCP (connection-bound) traffic in sockets, and each socket data structure contains its own sequentially-incrementing counter. Any time a thread is setting up a packet as part of a TCP connection, it's responding to a system call that already provided the socket structure as context. So there's no cost to "locating" the connection counter via some kind of hash function, like there is in per-destination and per-bucket selection. Moreover, there's essentially no situation in which multiple cores would be contending over the same socket simultaneously. So from the perspective of our performance benchmark, the complexity of requesting a per-connection IPID is as simple as standing up a `uint16_t` and incrementing it.

Per-Destination. We model our per-destination implementation based on the Windows reverse-engineering efforts of Klein [44], maintaining a hash table of (16-bit counter, 32-bit timestamp) pairs indexed by source and destination IP address pairs. Following the "purge sequences" that Windows Server (v1904) uses to limit its hash table sizes, we fix a purge threshold of $r = 2^{15}$ entries [45]. Purge sequences are initiated and stale entries are removed as described in Section 2.3. Because adding, removing, and updating destination counters are not thread-safe, we lock the entire hash table for each IPID request. Klein does not report on how Windows handles concurrency [44], and this appears to the be only solution that eliminates all race conditions.

Algorithm 1 Benchmark Implementations of Multi-Core IPID Selection Methods

---

1: Let c be a 16-bit atomic global counter.
2: **function** GLOBALLYINCREMENTING( )
3:     Atomically fetch-and-add $v \leftarrow c$ and $c \leftarrow c + 1$.
4:     **return** $v + 1$.

5: Let x be a local 16-bit counter. Note that this estimates per-connection's performance, not its IPIDs.
6: **function** PERCONNECTION( )
7:     Instantiate a counter x and set $x \leftarrow x + 1$.
8:     **return** x.

9: Let ℓ be a lock, h be a hash table of (16-bit destination counter, 32-bit last access time) pairs indexed by $(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}})$ pairs, and t_{purge} be the timestamp of the last purge sequence.
10: **function** PERDESTINATION($\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}}$)
11:     Lock ℓ and get the current time t_{now}.
12:     **if** $t_{\text{now}} - t_{\text{purge}} \geq 0.5$ s **then** initiate a purge sequence (see Section 2.3) and update $t_{\text{purge}} \leftarrow t_{\text{now}}$.
13:     **if** $(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}}) \notin h$ **then** insert $h[(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}})] \leftarrow (\text{random IPID}, t_{\text{now}})$.
14:     **else** $h[(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}})] \leftarrow (\text{counter}(h[(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}})]) + 1, t_{\text{now}})$.
15:     Store $v \leftarrow \text{counter}(h[(\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}})])$.
16:     Unlock ℓ and **return** v.

17: Let ℓ be an array of r bucket locks, h be an array of r 16-bit bucket counters, t be an array of r 32-bit last access times, and key be a randomly generated 128-bit hash key.
18: **function** PERBUCKETMUTEX($\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}}, \text{prot\_num}$)
19:     Compute $j \leftarrow \text{SIPHASH}(\text{dst}_{\text{ipaddr}}, \text{src}_{\text{ipaddr}}, \text{prot\_num}, key) \bmod r$.
20:     Lock $\ell[j]$.
21:     Get the current time t_{now} and generate a random increment $inc \leftarrow \mathcal{U}(1, \max\{1, t_{\text{now}} - t[j]\})$.
22:     Update $h[j] \leftarrow h[j] + inc$ and $t[j] \leftarrow t_{\text{now}}$.
23:     Store $v \leftarrow h[j]$.
24:     Unlock $\ell[j]$ and **return** v.

25: Let h be an array of r 16-bit atomic bucket counters, t be an array of r 32-bit atomic last access times, and key be a randomly generated 128-bit hash key.
26: **function** PERBUCKETLINUX($\text{src}_{\text{ipaddr}}, \text{dst}_{\text{ipaddr}}, \text{prot\_num}$)
27:     Compute $j \leftarrow \text{SIPHASH}(\text{dst}_{\text{ipaddr}}, \text{src}_{\text{ipaddr}}, \text{prot\_num}, key) \bmod r$.
28:     Atomically swap the current time $t_{\text{now}} \rightarrow t[j]$ with the previous time $t_{\text{old}} \leftarrow t[j]$.
29:     Generate a random increment $inc \leftarrow \mathcal{U}(1, \max\{1, t_{\text{now}} - t_{\text{old}}\})$.
30:     Atomically fetch-and-add $v \leftarrow h[j]$ and $h[j] \leftarrow h[j] + inc$.
31:     **return** $v + inc$.

32: Let ℓ be a lock, q be a queue of the last k IPIDs, and m be an array of 2^{16} membership tracking bits.
33: **function** PRNGSEARCHABLEQUEUE( )
34:     Lock ℓ.
35:     Repeatedly generate IPID values $v \leftarrow \mathcal{U}(0, 2^{16} - 1)$ until $v \neq 0$ and $m[v] = \text{FALSE}$.
36:     **if** $|q| = k$ **then** dequeue the last IPID u from q and set $m[u] = \text{FALSE}$.
37:     Enqueue v into q and set $m[v] = \text{TRUE}$.
38:     Unlock ℓ and **return** v.

39: Let ℓ be a lock, p be the permutation of IPIDs, i be the head index, and k be the number of reserved IPIDs.
40: **function** PRNGKNUTHSHUFFLE( )
41:     Lock ℓ.
42:     Let $v \leftarrow p[i]$ be the next IPID in the permutation.
43:     Swap $p[i]$ with $p[i - \mathcal{U}(0, 2^{16} - k - 1)] \bmod 2^{16}$, then increment i.
44:     **if** $v = 0$ **then** go to Line 42.
45:     **else** unlock ℓ and **return** v.

Per-Bucket (Linux). For each of the *r* buckets, we maintain a counter and a last access timestamp. Following Linux's implementation, we locate the bucket for a given packet using the SipHash-2-4 hash function on the packet's destination and source IP addresses, the protocol number, and a randomly generated 128-bit hash key. A stochastic increment is generated using the difference between the current time and the last access timestamp, after which the increment is applied and the timestamp is updated. Because multiple packets may simultaneously hash to the same bucket, some concurrency control is needed. Unlike in per-destination selection, however, multiple buckets can be accessed concurrently, avoiding the need for a global lock. We benchmarked two versions of per-bucket's concurrency control: one which protects each bucket with a lock, and another that follows Linux's actual implementation in making each bucket's counter and last access timestamp individually atomic. The latter allows a race condition where concurrent accesses to the same bucket could generate stochastic increments based on the same last access timestep and then atomically apply them to the same bucket counter. This does not necessarily pose significant issues for correctness or security, but may change the statistical properties of the IPID sequences slightly.

PRNG-based Methods. Our implementation of PRNG selection with a searchable queue directly follows FreeBSD's implementation, and our implementation of the iterated Knuth shuffle follows OpenBSD's. Both of these methods coordinate multiple instructions that must be made atomically: e.g., testing membership of a new IPID in the queue and then enqueuing it, or swapping two IPIDs in the permutation before advancing the permutation's start index. To eliminate race conditions, we lock access to the associated data structures for each IPID request. Our implementation of pure PRNG selection follows the macOS/XNU implementation, first reducing a 64-bit salt to a 16-bit salt, and then applying this reduced salt to an IPID chosen uniformly at random from among all 2^{16} values. Since this method does not keep state, concurrency issues can be sidestepped by giving each thread its own random number generator.

B.2 Benchmark Hardware

As explained in Section 4.4, we ran our primary benchmark on a 128-core (dual-socket AMD EPYC 7713 Zen3), 512 GiB node on Arizona State University's Sol supercomputer [38], yielding the results shown in Fig. 8. Since our benchmark depends heavily on CPU contention resolution mechanisms that may vary by CPU architecture, we ran a secondary benchmark on a 64-core (single-socket Intel Xeon Silver 4216), 256 GiB machine for the sake of comparison. The results, shown in Fig. 11, are consistent with those from the AMD CPU. Per-connection and pure PRNG selection remain the highest-performing methods since they have no contention resolution to perform; among the remaining methods that perform contention resolution, globally incrementing selection is once again the best. The only minor difference is that globally incrementing and per-bucket selection appear to scale slightly better on the Intel CPU than on the AMD CPU for 16 or more cores.
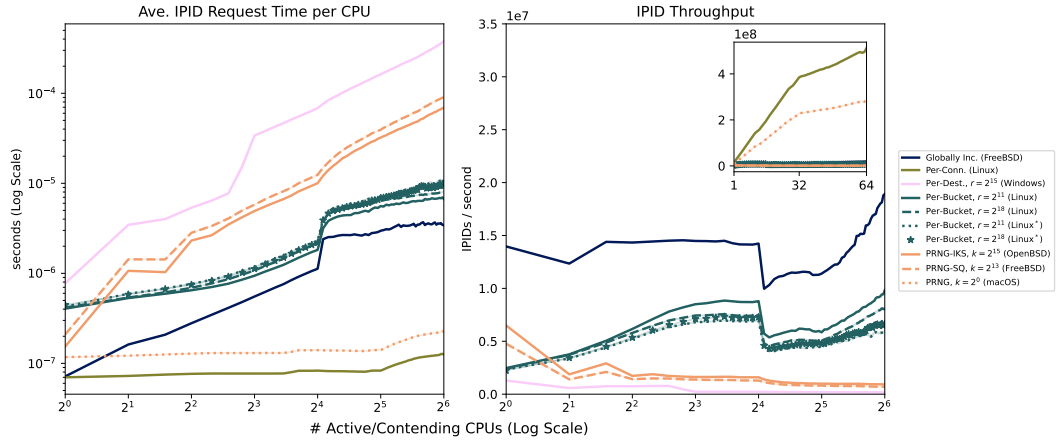
Fig. 11. *IPID Selection Methods' Time Complexities (Intel).* Analogous to Fig. 8, but using a 64-core Intel CPU.