Detecting Forged TCP Reset Packets

Nicholas Weaver
ICSI
nweaver@icsi.berkeley.edu

Robin Sommer
ICSI & LBNL
robin@icir.org

Vern Paxson
ICSI and UC Berkeley
vern@icir.org

Abstract

Several off-the-shelf products enable network operators to enforce usage restrictions by actively terminating connections when deemed undesirable. While the spectrum of their application is large—from ISPs limiting the usage of P2P applications to the "Great Firewall of China"—many of these systems implement the same approach to disrupt the communication: they inject artificial TCP Reset (RST) packets into the network, causing the endpoints to shut down communication upon receipt. In this work, we study the characteristics of packets injected by such traffic control devices. We show that by exploiting the race-conditions that out-of-band devices inevitably face, we not only can detect the interference but often also fingerprint the specific device in use. We develop an efficient injection detector and demonstrate its effectiveness by identifying a range of disruptive activity seen in traces from four different sites, including termination of P2P connections, anti-spam and anti-virus mechanisms, and the finding that China's "Great Firewall" has multiple components, sometimes apparently operating without coordination. We also find a number of sources of idiosyncratic connection termination that do not *reflect third-party traffic disruption, including NATs, load-balancers, and spam bots. In general, our findings highlight that* (i) *Internet traffic faces a wide range of control devices using injected RST packets, and* (ii) *to reliably detect RST injection while avoiding misidentification of other types of activity requires significant care.*

1 Introduction

Arguments tend to become heated when network operators restrict their users' communication by actively interfering with traffic. Recently, when Comcast was accused of interrupting their customers' BitTorrent connections [11], the public debate eventually even led to a high-profile FCC hearing on the legality of their practice [13]. One consequence of such uproar is that network operators may decline to openly inform customers about active measures they deploy, leaving users with only speculation about the cause for connections terminating without apparent reason—and sometimes users then *wrongly* accuse their ISP [26].

A different form of such active traffic interference comes not from ISPs but governments. For example, the "Great Firewall of China" censors Internet communication by terminating connections that relate to transfer of information deemed undesirable by the Chinese government [12].

Faced with uncertainty about the presence and degree of active interference, a natural question arises: to what degree can we *detect* when a network actively disrupts communication? In this study, we pursue answering this question. We focus on a specific, commonly deployed method to terminate an active connection on demand, namely the injection of forged TCP Reset (RST) packets into TCP flows, which manipulates the involved endpoints into shutting down their communication. While this method has been well-known for years, it recently gained traction with several companies now offering such functionality in high-performance, off-the-shelf products.

A crucial observation about RST injectors is their *out-of-band* operation. They can modify neither timing nor content of any packets sent by end-hosts (if they could, they could control traffic by simply dropping any further packets). Therefore, such injectors face *race conditions*: between the time when they inject RSTs until the endpoints receives these, the TCP connection state can *change* due to the transmission or reception of additional legitimate packets. These changes can delay the connection termination or even render it ineffective.

In this work we exploit these race-conditions to *identify* instances of injected RSTs via passive monitoring. We develop a set of tests for a number of relevant situations and combine them into a detector that can operate on both traces and real-time on live traffic. In addition, we find that many real-world injectors exhibit idiosyncratic peculiarities in the specifics of how they craft the RST packets, enabling us to develop *injector fingerprints* that identify which specific device is deployed on a given network path.

We have designed our detector to operate in a conservative fashion: it only reports RSTs that with high probability correspond to external injection, preferring false negatives over false positives. This trade-off is crucial because, as we see during our evaluation, regular network devices can also create unusual situations that a passive observer could misinterpret as a sign of injection. When carefully examining our datasets, we indeed discover anomalous RSTs sent by NATs, load-balancers, PlanetLab hosts, and buggy TCP implementations of spam bots. Thus, it is valuable to not only detect a RST as suspicious, but to also develop fingerprints in an attempt to classify an injected RST's source. Finally, while in this work we do not aim to take a position regarding the legitimacy of active traffic interference, we note that we observe evidence of anti-spam and virus blockers that also use RST-injection to block malicious traffic.

We structure the remainder of this paper as follows. In Section 2 we cover related work on known sources of injected RST packets. Section 3 discusses the principles of out-of-band flow blocking. Section 4 discusses the freedoms involved in creating RST packets, both from the end host and for an injector. Section 5 presents our detector for anomalous packets commonly generated by RST injectors. Section 6 introduces the four datasets we used in our evaluation, and Section 7 discusses the injectors we were able to find and fingerprint, as well as several types of anomalous RSTs not caused by packet injection. We conclude in Section 8.

2 Related Work

A study by Arlitt and Williamson [1] shows that RSTs are surprisingly common on the Internet. They examined a year of SYN/FIN/RST packets from the University of Calgary's border and found that roughly 15% of all TCP flows were terminated by a RST packet *after* payload had already been sent in at least one direction. The reset rate was even higher for HTTP traffic, with 22% of the flows terminated by a client-side RST, and 3% by a server-side RST. To understand these surprisingly high numbers, the authors evaluated different combinations of Web servers and clients to determine when they generate RSTs instead of normal FIN shutdowns. Among other effects, they discovered Web servers closing idle connections with RST packets as well as browsers consistently terminating persistent connections with RSTs.

Packet injection is a well known technique employed by network intrusion detection systems (NIDS) to terminate malicious connections. Snort's [24] `sp_response` and `sp_response2` plugins support RST and ICMP injection. The Bro NIDS [19] likewise comes a with a tool to inject RST packets. Song's tcpkill [25] is a stand-alone utility for the same purpose. We discuss the operation of these tools in Appendix B.

A well-known deployment of RST injectors is the "Great Firewall of China", which terminates Internet connections deemed undesirable by the Chinese government [12]. Clayton et al. [5] observe that the "Great Firewall" sends sequences of RST packets with TCP sequence numbers increasing by 1460 with each packet,[1] apparently to compensate for potential further data having arrived at the destination in the meantime, as discussed below. They also report that the RSTs have IP TTLs that differ from other packets from the purported source address. Once a host pair has had a connection terminated, the "Great Firewall" then sends individual RSTs for each newly initiated connection to maintain the block. As a counter-measure, the authors propose to ignore RST packets with wildly different TTLs. However, as developed in our study, we do not find this a practical mitigation technique, as similar wildly different TTLs arise in normal traffic (see Appendix C). Crandall et al. [8] spent considerable effort mapping the "Great Firewall", including determining the first point where filtering occurs by sending probes with different TTLs, and developing keyword maps of the detector's sensitivity. Independent of the detailed functioning of the "Great Firewall", Fallows [12] argues that it does not need to be technically perfect to reach its goal; rather it suffices to make access to external information enough of nuisance to spur people to prefer using resources within China's borders.

A recent controversial use of RST injection is restricting peer-to-peer (P2P) traffic as practiced by multiple ISPs, particularly to block bulk transfers such as those of BitTorrent [2]. Extensive publicity surrounded Comcast's use of this technique [11], leading to significant debate and multiple (somewhat ad hoc) studies. It can prove difficult to conduct such investigations in a sound fashion. One study, since retracted, claimed detection of RST injection that in fact occurred due to an artifact of the local NAT reacting to a large number of distinct flows [26]. Vuse, based on simply counting the total number of received RST packets seen by a client, claimed that AT&T performs RST injection without regard to their context [27]; AT&T denied these allegations [4]. The EFF has initiated a "Test Your ISP" project [10] with the goal to develop information and software tools that allow customers to examine their Internet connections for active interference. So far the two tools released in this context are *pcapdiff*, which compares two packet traces of the same communication captured at different locations for telltale differences, and *Switzerland*, a higher-level tool that automates the comparison process by utilizing a central server. In both cases, only if both sides of a flow are operating the tool, injected RSTs and other changes will be detected. Dischinger and colleagues developed a Java applet for volunteers to run which imitates

[1] 1460 is a common maximal TCP payload, based on 1500-byte Ethernet payloads minus 40 bytes of TCP/IP headers.

BitTorrent traffic [9]. Although they used a very different method, many of their results agree with ours in terms of detecting individual ISPs, including Cox, Comcast, and StarHub.

A somewhat different RST injection attack than those we consider in this study is *blind* RST injection. While its goal is the same—externally shutting down a connection using forged traffic—here attackers cannot observe the connection's packets. As such, they lack sufficient information to craft in-sequence RST packets, but they can still carry out brute-force attacks by sending many RSTs with different sequence numbers (abetted by guessing likely values of some fields), hoping to hit the target's TCP window with at least one. As Watson [28] shows, such an attack can be successful within a few minutes using a DSL line. The threat of such attacks disrupting Internet routing lead to the development of the TCP MD5 signature option [14], and [21] proposes requiring RSTs to exactly match the current sequence point.

3 Out-of-Band Flow Blocking

In this section we summarize approaches to block communication deemed undesirable. We assume use of a traffic monitor that inspects TCP flows for violations of a network's policy; it instructs a (generally) independent *connection terminator* to stop those identified. Such policy decisions can for example be taken based on security policy (e.g., by an IDS), access restrictions (e.g., China's "Great Firewall") or for traffic management purposes (e.g., Comcast's BitTorrent policy). The main difference of such a monitor/terminator setup compared to a traditional firewall is that typically all flows are initially allowed through ("default allow"), with potential blocking decisions taken only later if a connection is found to violate policy.

Devices to interrupt communication can operate either *inline* or *out-of-band*. For inline devices, blocking undesirable connections is easy: once the drop decision is made, the device simply ceases to forward (i.e., drops) all subsequent packets associated with the flows. However, inline operation also introduces new points of failure and can easily become a performance bottleneck. Consequently, many operators prefer out-of-band devices operating on a *copy* of the traffic stream (e.g., received via an optical splitter), which does not impact the network's principle operation when stressed or upon failure. This may be true even when devices support inline operation, such as the Sandvine tool used by Comcast [6].

Since out-of-path devices cannot directly block undesirable traffic, they must resort to indirect mechanisms to terminate flows, of which several exist: *(i)* instruct an existing in-path device, such as router, to block the flow (ACL injection); *(ii)* insert bogus TCP data packets to desynchronize the endpoints' TCP stacks (this can however lead to "storms" of packets between the endpoints that consume considerable network resources [15]); *(iii)* inject forged TCP FIN packets into the flow, one for each direction; and *(iv)* injecting forged RST packets instead of FINs, which has the advantage of requiring only one endpoint to accept a packet, and runs less risk of desynchronization storms.

In this study, we focus on the last of these, injection of forged RST packets, a method commonly used today (e.g., it is deployed by the "Great Firewall" as well as by Comcast's P2P disrupter). More broadly, however, the principles underlying our techniques—in particular, the insight that injection based on passive monitoring will face race conditions due to delays in the packet creation process—should apply to other forms of injection, including TCP FIN packets and spoofed DNS replies.

4 Properties of RST Packets

We now explore how benign, end-host initiated RSTs should appear versus how injectors can craft their packets. (Not surprisingly, we find end-hosts do not always behave like they "should", however, per Section 7.2.) According to RFC 793 [20], an end-host should sent a TCP RST packet when it either aborts (prematurely terminates) an existing connection, or when it receives a TCP packet (other than an initial SYN or a RST) that does not correspond to an active connection, which includes connections already aborted. Once an end-host has sent a RST for a connection, it should not send further data packets. It can however send more RSTs in response to continued traffic from the other side of the connection.[2]

The crucial field in a RST is its *sequence number*, which must be chosen correctly for the packet to be accepted by the destination. Per the RFC, when aborting a connection the sender should send an *in-sequence* RST, i.e., set the sequence number to the next available octet in sequence space if terminating an active connection. If the host is responding to a packet received for an inactive or already closed connection, the RST's sequence number should reflect the ACK field in the eliciting packet (or zero, if ACK was not set). Thus, the first RST packet sent should not have a sequence number lower than a previous data packet—although subsequent RST packets, responding to ACKs for data sent earlier in the sequence space, may use a lower sequence number.

The RFC however also specifies that *receivers* should treat arriving RSTs liberally: any *in-window* sequence number is considered acceptable because data packets preceding the RST may have been lost. Yet not all TCP stacks follow

[2] This is another reason why TCP RSTs, rather than FINs, are preferable for terminating connections. With a FIN, a host may accept a FIN but still send data in a half-open state, while a host that accepts a RST will neither accept nor send subsequent data on that connection.

this advice. Some are very lax and accept RSTs outside of the window; others are strict and require the sequence number to be exactly in-sequence, ignoring other values within the window (which prevents blind RST injection attacks [21]). Figure 4 of [23] summarizes the behavior of numerous systems.

An injector might attempt to exploit the standard's advice by sending RSTs with multiple sequence numbers, with the additional sequence numbers deliberately picked higher than the current sequence point in order to counter the race-condition of further data packets being already in flight (see Section 5). We do not expect to see such behavior from benign end-hosts, as this would require the end host sending RSTs that don't correspond to any data packets sent or received.

Other fields of the IP and TCP header are less crucial for a RST packet, and an injector has therefore considerable freedom in choosing them. If their values however divert from characteristics exhibited by the purported endpoint, a possibility for fingerprinting or detection arises.

Four significant header fields not checked for correctness when receiving a RST packet are other TCP flags, TCP ACK number, IPID and TTL. We would however expect an end-system to set these in a *consistent* fashion. According to our analysis, common choices for the ACK number are zero, the current sequence point, and an ACK number correctly acknowledging received data. The IPID is often zero, or incremented in consistent steps for subsequent packets. We might also expect that the TTL should not vary significantly across packets from the same source.

For all three of these fields, an injector can in principle pick arbitrary values for its forged RSTs. Looking for inconsistencies thus would appear to offer a means to spot injectors that do not try to evade detection. However, as we report in Appendix C, both IPID and TTL are highly volatile even for normal RST traffic. Thus, they are not suitable by themselves for detecting injected RSTs, but do prove useful in constructing fingerprints for individual RST injectors.

Another feature to look at is *payload*. While RST packets can carry data payloads (for diagnostic messages—*not* part of the regular bytestream), most commonly they do not. The forged RSTs we have observed are usually also empty, and therefore the presence of payload does not provide a suitable feature for detection. As we show in Section 7.1.7, there are however sources that insert readable messages into RST packets.

Finally, the *timing* of RST packets is important to consider as well. The gap between a RST and the packet preceding it can vary widely for end-host generated RSTs. For example, Web browsers often abort connections within milliseconds, while RSTs triggered by state timeouts are preceded by a substantial interval of non-activity. An injector does not have this freedom: the longer it takes it to inject the RST, the higher the likelihood that further packets are transmitted between the endpoints, rendering the termination ineffective. Therefore, in our injection detector we focus on RSTs occurring in short succession to the preceding packets.

5 Detection Toolbox

We now develop a set of detectors for abnormal situations that active, out-of-band RST injection can cause. As our discussion in Section 4 shows, due to the large degree of freedom an injector has when building a RST packet, a passive observer cannot always reliably differentiate between injected RSTs and normal end-host/network behavior. Therefore, when building our toolbox of tests, we do not strive for comprehensive coverage of all the ways in which an injected RST packet can show up at our monitoring point. We rather pick cases in which injection causes artifacts sufficiently distinct from normal end-host traffic to warrant further inspection. As we later show in Section 7.1, our set of detectors is indeed able to identify a wide spectrum of active interference.

Each of our detectors targets a specific situation that is likely to indicate the presence of one or more injected RST packets. We assume that injectors will send at least one RST to each endpoint of the connection to be terminated, which is nearly all injectors known to us work (the exception is tcpkill [25]). In the following we describe the detectors informally and refer to Appendix A for their precise definitions.

We start with two detectors, `RST_SEQ_DATA` and `DATA_SEQ_RST`, which target two race conditions that any out-of-path RST injector inevitably faces:

- `RST_SEQ_DATA`: One race condition occurs between the time when an injector sees a data packet that triggers its decision to terminate the connection, and the time when the injector sends out the fake RST packet. During this interval, further packets from the sender may pass the injector's observation point. If this happens we will observe that the RST packet is "out of sequence", with the receiver observing a sequence number less than the preceding data packet would suggest, a condition we detect as `RST_SEQ_DATA`. Most receivers will likewise consider the RST to be out-of-sequence and therefore ignore it. As data packets are often sent quickly back-to-back, we expect this situation to occur frequently when an injector is in use. In the absence of injection, however, it should not occur during normal TCP operation, other than in quite peculiar situations.

- `DATA_SEQ_RST`: Another race condition occurs when at the time the RST is injected, further packets are now already in flight, or will be sent shortly later, because the injector cannot stop the sender quickly enough. In these cases, the receiver will see further data packets from the sender *after* it has already received the RST. Our detector DATA_SEQ_RST triggers for such situations by looking for data packets having a larger sequence number than indicated by a previously arriving RST packet. Again, this situation should in general not occur during normal end-host communication.

These race conditions do not have to occur. In particular, RST_SEQ_DATA race conditions depend upon the reaction time of the injector—whether it can make a decision and generate a RST packet before the next packet passes the injector. Thus, the prevalence of this race condition may depend on the injector's implementation and current load. The DATA_SEQ_RST race depends more on network topology. If the injector is far from the end-host, it is more likely that there will be a subsequent in-flight packet.

Our third detector triggers when it sees a common counter-measure many injectors take: sending multiple RSTs instead of just one. Without this countermeasure, a conforming TCP stack would ignore the RST packet when a RST_SEQ_DATA race occurs.

- RST_SEQ_CHANGE: By quickly sending multiple RSTs with increasing sequence numbers, an injector can increase the likelihood of getting at least one of them through. It however faces the dilemma of having to pick a higher sequence number without knowing what the source will sent, and therefore might guess a value higher than the maximum sequence number the receiver will have seen at the time the RST arrives. The RST_SEQ_CHANGE detector leverages this observation by looking for back-to-pack pairs of RSTs in which the second RST has a sequence number higher than the first, and that exceeds the current maximum sequence number. A standard compliant TCP stack should never send such a packet because its RSTs should either be in sequence with the data (so at the maximum sequence number) or in response to packets from the other side (which should have an ACK field less than the maximum sequence number sent).

The RST_SEQ_CHANGE detector does not depend on a race condition. Rather, it detects a natural consequence of constructing a robust RST injector. Thus, our detector is not guaranteed to detect injectors that are not robust to the RST_SEQ_DATA race condition, but will detect injectors that send multiple packets to avoid the race condition.

Finally, we add three more detectors to our toolbox which, even though they are not *clear* indicators for the presence of an active injector, trigger for RST traffic that is sufficiently odd to warrant further inspection:

- RST_ACK_CHANGE: Detects RSTs with seemingly nonsensical ACK numbers. Specifically, the detector looks for pairs of RSTs in which the second RST's ACK number differs from its predecessor and does not lie within the range of sequence numbers seen from the data sender. Although not a necessary feature for injected RSTs, we have observed that some injectors incorrectly increment the ACK rather than the SEQ field when sending multiple packets.

- SYN_RST: Detects initial SYNs immediately followed by a RST in the same direction. While this behavior can occur benignly for some applications (e.g., Web browsers), it can be an indicator of active interference for others.

- SYN_ACK_RST: Detects initial SYN/ACKs immediately followed by a RST in the same direction with no intervening packet. Similar to SYN_RST, this can be an indicator of RST injection. We however also see it with servers making a decision to accept a connection only after their TCP stack has already acknowledged the initial SYN (e.g., because load-monitoring finds the server's load too high to accept new requests, or due to consulting an SMTP blacklist).

Finally, for our detectors we need to select values for two parameters (T_1 and T_2 in Appendix A). The first of these governs the maximum delay an injector can exhibit in issuing its response to traffic, for which we chose 2 sec as sufficient for a very slow injector even on a very slow link. RSTs with larger delays likely reflect state management or sender-side bugs rather than injection. The second parameter bounds the delay for termination of connections during the establishment phase (for the SYN_RST and SYN_ACK_RST detectors). Here we chose 0.1 sec, because such decisions should be quick for an injector to make (since only inspection of header information can come into play).

We implement our detector in Click [16], aiming for high performance when running on large traffic streams such as campus borders. To keep memory management efficient and simple, we use a fixed cache to track active flows, rather than dynamically allocated tables. We provision the cache with 256K entries and 32-way associativity with LRU replacement. Bad evictions from this cache lead to missed alerts rather than false positives; we checked for such evictions when running on particularly large *UCB* traces and did not record any that would have resulted in loss of accuracy. To enable further analysis, we couple the detector to a 500K-packet buffer to extract context surrounding possible detections.

We insert all detections into a database, including packet headers for the alerting packet, up to 200 prior and 100 subsequent packets, and payloads of any RST packets. This provides us with significant context around the alert to develop and evaluate fingerprints of injectors. We also store in the database the fully qualified reverse-lookup (PTR) for the IP addresses, excluding the actual hostname (thus `foo.bar.baz.com` is recorded as `bar.baz.com`), as well as the nation, state, and city lookup results from the GeoLite City GeoIP database [17]. To enable others to run our detector, it optionally can anonymize the IP addresses and hostnames.

6 Datasets

We used the datasets from four institutions for our study:

International Computer Science Institute: We ran a prototype of our detector at *ICSI* from January 23rd, 2008 until May 1st, running on all TCP traffic other than SSH. This detector was used to guide a "hosts-of-interest" selection, capturing all traffic between any two hosts generating an alert for later analysis. During the measurement period the detector was not static, but received several improvements. It initially only detected `DATA_SEQ_RST` and `RST_SEQ_DATA` anomalies, but later ran the entire complement of alerts. Thus, we cannot use this data to gauge the overall presence of injected packets, but because it has extensive context it allows detailed investigation of individual activity.

UC Berkeley: We captured the *UCB* trace using an experimental intrusion detection cluster that receives traffic from the campus' two border routers. As the routers aggregate traffic onto a single 1 Gbps SPAN port, this environment can saturate during traffic peaks. We captured data representing 40% of the total border traffic, except for data involving *UCB*'s PlanetLab nodes. The monitoring setup receives a subselection of the flows from the SPAN port; in most cases, both halves of each flow, but in some cases only a single side. These latter do not hinder our analysis except that we suppress the `RST_ACK_CHANGE` alert, and the `RST_SEQ_CHANGE` alert does not check the ACK value.

This trace ran for 19 hours starting at 2PM, April 21, 2008, capturing 5.2 Gpkts and 73M TCP flows. Excluding backscatter and partially created flows, the trace contains 30.2M TCP flows.

In evaluating this trace, we also verified that our caching was not causing problematic evictions: we experienced no evictions from our data structure's caches for data less than 4 sec old. Thus, our caching-based structure did not cause us to miss alerts. However, the limited buffer of 500K packets did cause us to lose significant context for the alerts. At worst, the buffer only held 7 sec of associated traffic, limiting the context around each alert for further analysis.

Columbia University: The *Columbia* trace consists of a day captured at the border of the institute's Computer Science Department, excluding PlanetLab servers. We do not have packet counts for this trace.

George Mason University: The *GMU* trace consists of 5 hr of traffic captured at the campus border, totaling approximately 70 GB. This trace was processed live rather than offline.

For all traces, we excluded `SYN_RST` alerts for ports 80, 113, and 443, and `SYN_ACK_RST` alerts for ports 25, 80, and 443, in both cases due to there being a large number of benign causes for the alerts. (For example, Facebook's HTTP servers generated a large number of `SYN_ACK_RST` alerts in the *UCB* data.) One source of `SYN_RST` alerts on port 80 and 443 comes from users hitting the "Stop" button on their web browser. Alerts on port 113 arise from how some mail servers contact the "identification" service. `SYN_ACK_RST` alerts on port 25 can be due to mail server aborts, where the mail server accepts a connection and then checks a blacklist, while port 80 and 443 alerts appear due to high-load issues, where a Web server will initially accept a connection and then reject it due to its load policies.

Once all alerts, context, and data are loaded into the database, we were able to correlate between multiple alerts and develop fingerprints of individual RST injectors as well as benign sources. We developed these fingerprints through manual examination, looking for common patterns present in the alerts from the same and different IP addresses.

When we could fingerprint an injector or a non-injected source, we classified it as either a true detection or as non-injected (such as due to a misbehaving in-path device or a misconfigured TCP stack). In addition, as discussed later, we also find behavior that we deem as likely one or the other, but, because we could not determine a reliable fingerprint for it, we cannot precisely identify.

7 Results

We now present the results of our detector running on the datasets discussed in the previous section. We start with the kinds of injectors we were able to identify by their characteristic fingerprints, followed by a discussion of unexpected RSTs we observed that do *not* appear due to out-of-band injection. Table 1 summarizes the fingerprints we determined for different RST injectors, and Table 2 summarizes the alerts for these reported by the detector.

Counts in Table 2 reflect distinct IP addresses, not distinct flows. Any given address may have multiple flows that generate an alert, as systems may retry connections.

Identified Source	Signature
Identified Injector	
Sandvine	Multipacket: First Packet IPID += 4, second packet SEQ + 12503, IPID += 5
Bezeqint	Multipacket: Constant sequence, `RST_ACK_CHANGE`, IPID = 16448
Yournet	`SYN_RST`: Only on SMTP, TTL usually +3 to +5, unrelated IPID
Victoria	Multipacket: Sequence Increment 1500, IPID = 305, TTL += 38
IPID 256	Single packet: Usually less TTL, IPID = 256
IPID 64	Multipacket: IPID = 64, often sequence increment of 1460
IPID -26	Multipacket: First IPID -= 26, often sequence increment of 1460
SEQ 1460	Multipacket: Sequence increment always 1460
RAE	Single packet: Sets RST, ACK and ECN nonce sum (control bit 8)
Go Away	Single packet: Payload on RST of "`Go Away, We're Not Home`"
Optonline	Multipacket: No fingerprint, all activity from a single ISP
Identified Non-Injected Source	
SYN/RST 128	`SYN_RST` with RST TTL += 128
SYN/RST 65259	`SYN_RST` with RST IPID = 65259
0-Seq RST	Reset with SEQ = 0
IPID 0	IPID = 0, multiple RSTs, limited range
IPID 0 Solo	IPID = 0, spurious RST (often ignored)
Stale RST	RST belonging to a previous connection (port reuse)
Spambot SR	Spam source sending payload packets with SYN and RST flags
DNS `SYN_RST`	Normal DNS servers aborting connections at initiation

Table 1. Features for both identified RST injectors and identified non-injected sources.

7.1 Identified RST Injectors

By correlating the characteristics of RSTs across our datasets, we identified and fingerprinted a number of injectors that we believe our detector consistently identifies. We present these in Section 7.1.1–7.1.6 and then discuss in Section 7.1.7 additional cases that appear likely to reflect injection, yet for which we lack sufficient evidence to confirm that suspicion.

7.1.1 The Sandvine RST Injector

Comcast has publicly stated that they use RST injection to manage P2P traffic [11], and it has been reported that these devices were purchased from Sandvine [22, 6]. We examined all flows reported by our detector involving a Comcast host (as identified via reverse DNS lookups). Across the four sites, 90% (174 of 193) of the alerting sources have at least one alerting flow with a back-to-back pair of RSTs for which the second has a sequence number 12503 higher than the first and an IPID incremented by 1. Additionally, in 164 cases at least one of the alerting flows had the IPID of the first RST corresponding to that of the previously seen packet incremented by 4.

Given the consistency of these RSTs, we consider these features to be a fingerprint of the Sandvine injector. In the ICSI trace we observe 106 distinct Comcast IP addresses, 30 at *UCB*, 36 at *Columbia*, and 2 at *GMU* (top row of Table 2).

The Sandvine CTO subsequently indicated to us that the particular sequence number increment of 12503 represents a known bug in their tool, and that the intended increment was far smaller [3]. Incrementing the IPID by 4 does not have any fundamental reasons, since the only network mechanism sensitive to IPID (fragmentation) should not come into play. If the goal is to avoid repeating a previous IPID, selecting a value at random would work just as well, or using a larger increment.

Comcast's Use of Sandvine: We looked closer at Comcast's usage of RST injection to verify the company's public statements about its application of traffic management. At ICSI, we confirmed that RSTs reported for Comcast traffic indeed correspond to the usage of P2P software. Almost all of the Comcast alerts came in 4 bursts: 10 on February 9th, 23 on February 18th, 39 between March 8th and 10th, and 26 between April 22nd and 24th. Two bursts matched with reported instances of excessive bandwidth usage by local users running P2P software, and we verified that these remote hosts were communicating with the offending local systems. One of the solo alerts was also manually correlated with a user who forgot to turn off his BitTorrent transfer when entering ICSI's network. These alerts all reflected high TCP ports (> 1050), which fits with many forms of P2P software.

Identified Alert	Source	*ICSI*	*UCB*	*Columbia*	*GMU*
Identified Forged RSTs					
Sandvine	Comcast	106	30	36	2
Sandvine	Cox	35	262	3	0
Sandvine	Korea	1	50	4	0
Sandvine	Other	0	1	0	1
Bezeqint	Bezeq Int.	25	0	2	0
Yournet	yournet.ne.jp	29	0	0	0
Victoria	UVic.ca	1	0	0	0
IPID 256	Korea	9	90	16	0
IPID 256	Other	0	5	0	0
IPID 64	China	13	6	0	0
IPID -26	China	35	1	0	0
SEQ 1460	China	21	5	3	1
RAE	China	229	4,162	8	0
Total Identified		275	450	64	4
Possibly Forged RSTs					
Go Away	Various	3	5	0	0
Optonline	Optimum Online	12	0	0	0
Exact Multipacket	Various	7	11	2	0
Approx. Multipacket	Various	2	2	2	0
Total Identified		253	4,180	12	0
Identified Non-Injected RSTs					
SYN/RST 128	Various	98	36	2	0
SYN/RST 65259	Various	9	2	0	0
0-Seq Reset	Various	48	46	6	1
IPID 0	Various	17	35	19	0
IPID 0 Solo	Various	36	149	17	0
Stale RST	Various	36	72	3	1
Spambot SR	Various	11	1	0	0
DNS SYN/RST	Various	2	14	0	0
Total Identified		257	355	47	2
Likely Non-Injected RSTs					
Web Server	Various	17	134	1	0
SMTP `SYN_RST`	Various	61	54	0	0
Unknown `SYN_RST`	Various	38	172	10	0
Unknown `SYN_ACK_RST`	Various	5	321	14	0
Unknown `RST_ACK_CHANGE`	Various	74	97	32	5
Confused Multipacket	Various	18	36	7	1
Hanson	Hanson Infosystems	1	0	0	0
Total Identified		214	814	64	6
Total Unknowns		210	588	28	8
Total Sources		1,209	6,387	215	20

Table 2. Number of alerting source IP addresses and their classifications in each trace.

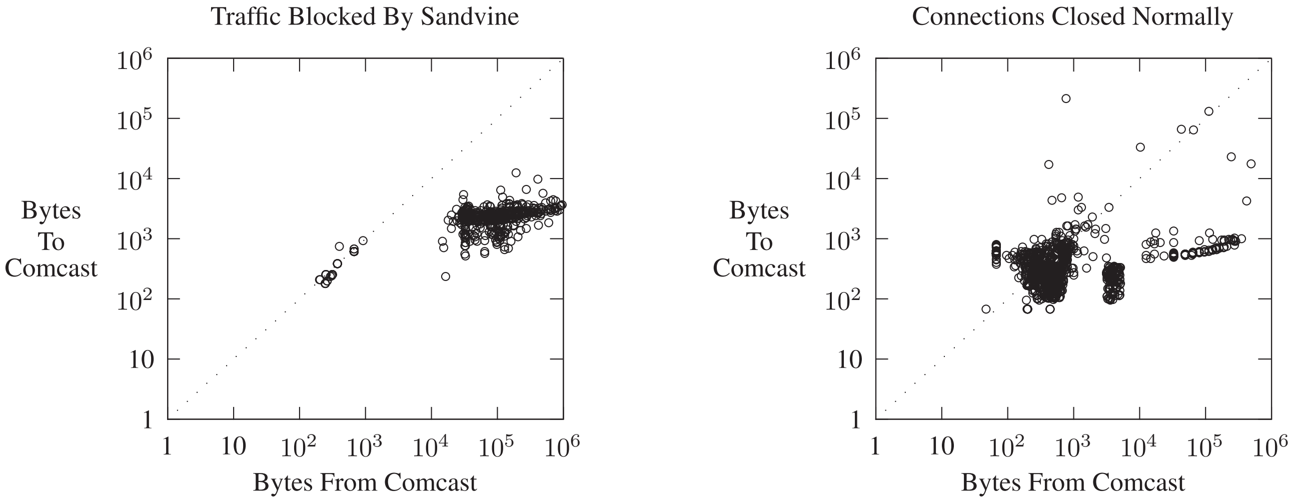

Figure 1. Bytes transferred and received to/from Comcast hosts communicating with ICSI. The left plot shows sizes for connections terminated by Sandvine-injected RST packets, with Comcast hosts identified based on hostnames. For comparison, the right plot shows sizes for all connections between the same ICSI hosts and Comcast hosts that were instead closed with a normal FIN handshake; here, we identified Comcast hosts based on WHOIS data for P2P traffic.

Comcast has stated that their P2P traffic management targets only uploads, i.e., Comcast users *sending* significant volumes to others [7]. To verify this, we estimated the data transferred by the affected flows in each direction before they were terminated. Figure 1 shows that terminated connections accorded with Comcast's statement—disruption mostly occurred on uploads from Comcast hosts, and did not occur on flows where the Comcast host received substantially more data than it sent. (The second plot shows that this pattern is not simply an artifact of regular communication patterns with Comcast hosts.) However, we also see that 7% of the affected flows did not transfer a significant volume of data in either direction before being blocked by an injected RST, suggesting that traffic upload is not the only discriminator in use. According to Sandvine, their software supports direct recognition that a client is acting as a BitTorrent seed by parsing BitTorrent messages [3].

Other Users of Sandvine: We have observed two other ISPs using the Sandvine injector: Cox Communications, and a Korean ISP. The former confirms a report by Topolski of Cox disrupting P2P traffic [18], and we have identified the tool in use as the same as deployed by Comcast. We have not identified the Korean ISP, but the fingerprint is clear. We also found one alerting source in each of two other traces, both geolocating to the USA, but without resolvable hostnames.

7.1.2 The BezeqInt Injector

Another injector consistently appears in traffic involving hosts from Bezeq International, an ISP belonging to the primary Israeli phone company. Like with Comcast hosts, we could confirm these cases as reflecting P2P usage at ICSI. Again, this is a multiple-packet injector. However, rather than changing the sequence number, for unknown reasons it increments the ACK number (based on the received packet window size, and without setting the ACK flag). It also *always* uses IPID 16448 and a differing TTL. These features appear due to either ease of implementation or bugs.

This injector operates more aggressively than Comcast's. Out of 30 flows blocked at ICSI, only two managed to exchange more than a few hundred bytes of data. For both of these flows, the data was almost exclusively sent from ICSI to the Bezeq International host.

7.1.3 The IPID 256 Injector

Another injection source we found is the "IPID 256" disruptor, an injector that uses a constant IPID of 256. We observe this injector primarily in hosts that geo-locate to Korea, along with some other Asian countries. Use of this injector appears unrelated to Korean use of the Sandvine injector. Again, this disruptor appears to target P2P traffic.

7.1.4 The Yournet Injector

At ICSI we observed 29 addresses that generated SYN_RST alerts, all from a single Japanese ISP, yournet.ne.jp. Each alert corresponds to SMTP traffic incoming to ICSI, representing 30% of all SMTP clients that exhibit only SYN_RST alerts. (There is also one RST_SEQ_CHANGE alert.) In this case we observe the TTL of the RST packet as usually 5 higher, and the IPID appears to have no relationship with the data IPID. Thus, it appears that yournet.ne.jp actively disrupts email delivery attempts, presumably in an attempt to control spam originated by bots.

7.1.5 The Victoria Injector

One peculiar host generated 96 alerts in ICSI traces during a 5-day period in April. From the traffic contents, this host appears to be a mail server that repeatedly attempts to deliver a "mail undeliverable" message triggered by the W32/MyDoom-O mail virus. The server never successfully transferred the message, with each attempt suffering interruption mid-transfer by a sequence of 10 RST packets. These RSTs always have IPID 305 and a TTL that is 38 higher than the data packet, and with sequence numbers increasing by 1500 per RST.

We speculate this traffic reflects an in-network "virus scanner" that heuristically (mis-)recognizes the bounce message as malicious. We attempted to contact *postmaster* and *security* at this site, but have not yet received a response.

7.1.6 The Chinese Injectors

We observe four distinct RST injectors that appear only in traffic with Chinese hosts. The "IPID 64" injector uses a constant IPID value of 64, and the "IPID -26" injector an IPID value 26 less than the preceding data packet. The "RAE" injector sets the RST flag, the ACK flag, and bit 8 of the TCP flags (ECN nonce sum). The "SEQ 1460" injector is a multipacket injector that increments the sequence number by 1460 regardless of the previous packet's size or apparent MTU; sets the ACK flag on the RST packet; and appears to choose an arbitrary IPID and TTL.

All of these injectors disrupt a variety of traffic, including email, Web, and P2P. The RAE injector is by far the most common, and apart from its strange use of the ECN nonce sum flag is hard to fingerprint. It is a single packet injector, so it does not generate clear RST_SEQ_CHANGE alerts. It often, but not always, takes its IPID from the previous packet. The injector's aggressiveness triggers SYN_RST and SYN_ACK_RST alerts as well as DATA_SEQ_RST and RST_SEQ_DATA alerts.

Sometimes multiple Chinese injectors operate *simultaneously*. For example, we observed an SMTP client communicating with the ICSI mail server that exhibits packets originating from both the SEQ 1460 and IPID 64 injectors, while a web server visited from *Columbia* manifests the IPID 64 injector, likely the SEQ 1460 injector (though an imperfect match), a RST seemingly generated by the end host and a RST apparently generated by the IPID -26 injector *whose IPID suggests that it was at least partially responding to the packet injected by the 1460 injector!* The only other apparent explanation is that our fingerprints are overly narrow, i.e., we have assigned two distinct fingerprints to the same device.

Of all 298 ICSI hosts classified as disrupted by one or more of the Chinese injectors, 102 hosts contain the fingerprints of two or more injectors. In general, the RAE injector appears independent of the other three (only two sources overlap), but the other three injectors appear to target similar, and sometimes the same, flows.

7.1.7 Likely RST Injectors

One interesting type of source sends RSTs with a payload of "Go Away, we're not home". The RST sequence numbers, although changing from packet to packet, never exceed the maximum-sent sequence, so we believe the source is either stateful or uses incoming ACKs to generate the sequence numbers; thus, we can only detect it when a RST_SEQ_DATA or DATA_SEQ_RST race condition occurs. We saw such sources from SBC/Pacific Bell (AT&T) as well as from two Mexican ISPs (prod-infinitum.com.mx and telnor.net). All alerts correlate with P2P activity. As these are not unique to just one ISP, and are too few to fully classify, we suspect the traffic could be generated by a non-ISP source—possibly end-system software.

It appears that Optimum Online, a division of Cablevision, terminates P2P flows as well. 12 sources at ICSI from this domain generate RST_SEQ_CHANGE alerts, which appear due to a multi-packet injector. The injector usually uses either the last packet's TCP payload size as the sequence number increment or twice this value. We were not able to generate a more precise fingerprint, and as we do not see any evidence of this injector in the other, more recent traces, we assume the practice may have been discontinued. Thus, we classify these only as a probable injector rather than a confirmed source.

Finally there is a group of systems that exhibit RST_SEQ_CHANGE alerts, either using an exact interval from the previous packet or a slightly different interval. We have been unable to classify these further, although some correspond to the StarHub network previously reported as blocking P2P by Dischinger et. al [9].

7.2 Apparently Legitimate but Unexpected RSTs

Our detector identifies anomalous RSTs, yet not all of them are due to injectors. We cross-checked the alerts using several strategies in order to assess those due to sources other than injection, including looking for RSTs sent by local hosts (for which we could obtain ground truth) and for external hosts known to not be subjected to traffic management. These may represent either in-path network devices with various bugs, or bugs in end-system TCP stacks, rather than packets injected by a separate traffic management/disruption system.

Just as RST injectors can show clear signatures, we can fingerprint some benign sources of unexpected RSTs as well. We discuss these cases first, followed by likely-non-injected RSTs for which we could not develop an effective signature.

7.2.1 Legitimate Resets With Fingerprint

Common SYN/RST Signatures: We see a large number of SYN_RST alerts with repeated signatures, including TTL 128 higher than the triggering SYN ("SYN/RST 128"), and a constant IPID of 65259 ("SYN/RST 65259"). As these signatures do not appear to have any geographic or ISP commonality, we consider them to reflect non-injected sources.

Common RST Signatures: Three other seemingly benign signatures are *(i)* RSTs with a sequence number of zero ("0-Seq RST"), *(ii)* sending multiple RSTs with IPID 0 within a limited sequence number range ("IPID 0"), and *(iii)* hosts that generate spurious RST_SEQ_DATA and DATA_SEQ_RST errors with a RST packet with IPID 0 in active flows ("IPID 0 Solo"). Traces of these appear quite peculiar; we suspect the behavior is due to middlebox or end-host bugs.

Stale RSTs: We observed a rare RST_SEQ_DATA alert generated by our institute's mail server. Further examination shows the cause: A system (presumably a spam bot) contacting the mail server first receives a SYN/ACK, prior to a blacklist check causing the server to terminate the connection. Several seconds later, the presumed spam bot connects again, using the *same* TCP source port (in violation of the TCP spec). This second SYN is acknowledged with a different sequence number, a few packets are exchanged, and then the mail server sends a TCP RST with the sequence number of the *first* flow, creating a completely out-of-sequence RST that trips the detector. We term this situation "Stale RST".

Spambot SYN/RST Bug: We observed non-injected RSTs due to an apparently buggy custom TCP stack in spam bots. These systems at first communicate normally, and then for unknown reasons generate an out-of-sequence packet with *both* SYN and RST flags set, and payload containing portions of a spam message.

DNS SYN/RST: We find that DNS servers can generate SYN_RST alerts on TCP communication, for unknown reasons. This appears to be benign activity caused by the end-system.

Planetlab: In an early test trace of *Columbia* traffic, we observed more than 300 distinct RST_SEQ_DATA and DATA_SEQ_RST alerts involving communication between Columbia's three PlanetLab nodes. We do not know the cause, but due to PlanetLab's experiment nature we excluded these.

7.2.2 Ambiguous Cases

HTTP Servers: Several domains, including Google and Yahoo, show rare DATA_SEQ_RST and RST_SEQ_DATA alerts with HTTP/HTTPS connections. We assume that these domains do not perform active traffic management via RST injection; manual examination did not reveal any apparent cause. We speculate this traffic is due to bugs or race conditions in HTTP load-balancers employed by these sites.

For example, the ICSI trace shows 18 instances of RST_SEQ_DATA alerts generated by `ad1.p1.vip.rm.sp1.yahoo.com`, where two MTU-sized data packets are sent followed by two RST packets. The first RST packet has a sequence equal to the start of the second data packet, and the second RST packet comes properly in sequence. Manually examining one of these connections shows an apparently normal request to one of Yahoo's ad servers. Google generates similar alerts, as well as DATA_SEQ_RST alerts.

We were not able to develop a fingerprint for such load-balancers, and thus consider Web servers that generate only RST_SEQ_DATA and DATA_SEQ_RST alerts as probably non-injected sources. However, the Web server of one particular site, `flightglobal.org`, does show a very distinct fingerprint. On an HTTP 302 ("`Temporarily moved`") error in a persistent connection, instead of sending a normal data packet it sends a TCP RST packet with the payload containing the HTTP "`Object Moved`" message. Not only does this not make sense, but the RST packet's sequence number equals that of the previous data packet: a RST_SEQ_DATA error.

SMTP `SYN_RST` alerts: Unless we find a significant clustering (e.g., the Yournet alerts in Section 7.1.4), SYN_RST alerts are so common from SMTP clients that we must treat them as non-injected sources.

Inefficacy of Some Tests: We find three of the alerts—RST_ACK_CHANGE, SYN_RST, and SYN_ACK_RST—nondefinitive on their own. We can sometimes correlate across alerts (such as the Japanese SMTP interference and the BezeqInt injector) to create a global picture or fingerprint, but in isolation these alerts do not provide convincing evidence of injection, so we consider them as likely not reflecting injected RSTs.

Confused Multipacket: Although RST_SEQ_CHANGE is an effective tool at fingerprinting injectors, we occasionally see obviously anomalous cases, where the second RST packet is very close (< 200) or very far away ($> 4x$) from the last data packet's position in the sequence space. We do not consider these as part of deliberate injection activity unless we can fingerprint them in some other manner (such as the Sandvine injector), because for deliberate injection the choice of increment would be ineffective and hence is puzzling.

Hanson Infosystems: We have observed a single SMTP server belonging to Hanson Systems that shows unusual behavior. It could be end-host software or it could be RST injection that is triggering on the message. This host generates RST_SEQ_DATA alerts when the ICSI mail server attempts to forward a user's spam to this site. The remote mail server issues a rejection message immediately followed by a RST packet with sequence equal to the previous packet's *starting* sequence, a RST_SEQ_DATA error.

NATs: One internal host at ICSI generated 30 alerts during operation, almost all RST_SEQ_DATA alerts, with one DATA_SEQ_RST alert. Investigation revealed that the source is not an end-host host but a NAT, so we suspect that the RSTs result from erroneous state expiration on the part of the NAT. (Erroneous because the connection was active at the time of termination.) We suspect that some addresses counted as "unknown" in Table 2 might likewise be due to NATs.

8 Conclusions

In this work we develop an efficient detector for forged TCP RST packets, as deployed for example by some ISPs to manage P2P traffic, as well as by the "Great Firewall of China" to censor communication deemed undesirable by the Chinese government. Our detector identifies injected RSTs by exploiting the race conditions that out-of-band injectors *fundamentally* face. We then further leverage the idiosyncratic peculiarities specific to many brands of injectors to fingerprint their particular type.

Using datasets from four network sites, our evaluation is able to confirm the use of RST injection by several ISPs. We also observe that multiple distinct injectors operate in China. As sometimes they are independently attempting to block the *same* connection, they may have been installed by local ISPs, independent of the "Great Firewall". In addition to traffic management and censoring, we also find RST injection used as a tool to counter spam and virus spreading.

Our study also shows the limits of passive monitoring to detect active traffic interference. The most fundamental limitation stems from likely benign in-network devices, often end-hosts, that produce abnormal effects similar to those observed when RSTs are injected. As regularly experienced by network researchers, the variety observed in network traffic includes many situations not covered by any RFC; in our case that means RSTs sent by buggy TCP stacks and misbehaving middle-boxes. We therefore designed our injection detector to operate in a conservative fashion, correlating several distinct properties to ensure reliable results. Our experiences also highlight the pitfalls one can encounter if assuming that peculiar RSTs necessarily reflect traffic control.

9 Acknowledgments

Special thanks to Angelos Keromytis, Gabriela Cretu, and Angelos Stavrou for running our detector at their institutions, Jim Mellander for suggesting desynchronization through packet injection, and Christian Kreibich for his feedback during this process.

This work was funded in part by NSF grants CNS-0722035 and ITR/ANI-0205519. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding source.

References

[1] Martin Arlitt and Carey Williamson. An Analysis of TCP Reset Behaviour on the Internet. *SIGCOMM Comput. Commun. Rev.*, 35(1):37–44, 2005.

[2] BitTorrent, www.bittorrent.com.

[3] Don Bowman. Private communication.

[4] Anne Broache. AT&T: We Don't Throttle P2P Traffic. http://news.cnet.com/8301-10784_3-9929158-7.html?part=rss&subj=news&tag=2547-1_3-0-5.

[5] Richard Clayton, Steven Murdoch, and Robert Watson. Ignoring the Great Firewall of China. In *6th Workshop on Privacy Enhancing Technologies*, 2006.

[6] Comcast Corporation: Description of Current Network Management Practices. http://downloads.comcast.net/docs/Attachment_A_Current_Practices.pdf.

[7] Comments of Comcast Comporation. http://fjallfoss.fcc.gov/prod/ecfs/retrieve.cgi?native_or_pdf=pdf&id_document=6519840991.

[8] Jedidiah R. Crandall, Daniel Zinn, Michael Byrd, Earl Barr, and Rich East. ConceptDoppler: A Weather Tracker for Internet Censorship. In *14th ACM Conference on Computer and Communication Security (CCS)*, 2007.

[9] Marcel Dischinger, Alan Mislove, Andreas Haeberlen, and Krishna P. Gummadi. Detecting BitTorrent Blocking. In *Internet Measurement Conference*, 2008.

[10] EFF "Test Your ISP" Project. `http://www.eff.org/testyourisp`.

[11] Ernesto. Comcast Throttles BitTorrent Traffic, Seeding Impossible. `http://torrentfreak.com/comcast-throttles-bittorrent-traffic-seeding-impossible`.

[12] James Fallows. The Connection Has Been Reset. *Atlantic Monthly*, March 2008.

[13] FCC Announces Public *En Banc* Hearing in Cambridge, MA on Broadband Network Management Practices, `http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-280194A1.pdf`.

[14] A. Heffernan. RFC 2385: Protection of BGP Sessions via the TCP MD5 Signature Option, 1998.

[15] Laurent Joncheray. A Simple Active Attack Against TCP. In *Proceedings of the 5th Usenix Security Symposium*, 1995.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[17] MaxMind GeoIP Address Location Technology. `http://www.maxmind.com/app/ip-location`.

[18] Cade Metz. Cox Pulls a Comcast with eDonkey, `http://www.theregister.co.uk/2007/11/20/cox_bagging_edonkey_swaps/`.

[19] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[20] Jon Postel. RFC 793 - Transmission Control Protocol, 1981.

[21] A. Ramaiah, R. Stewart, and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. Internet-Draft, `http://tools.ietf.org/html/draft-ietf-tcpm-tcpsecure-09`, 2008.

[22] Sandvine Intelligent Traffic Management, `http://www.sandvine.com/solutions/p2p_policy_mngmt.asp`.

[23] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[24] The Snort IDS, `www.snort.org`.

[25] Dug Song. dsniff tools. `http://www.monkey.org/~dugsong/dsniff`.

[26] Broadband Network Management, `http://systems.cs.colorado.edu/mediawiki/index.php/Broadband_Network_Management`.

[27] First Results from Vuze Network Monitoring Tool, `http://cache2.vuze.com/docs/internet_future/First_Results_from_Vuze_Network_Monitoring_Tool.pdf`.

[28] Paul A. Watson. Slipping in the Window: TCP Reset Attacks. Presentation at 2004 CanSecWest, `http://cansecwest.com`.

A The Complete Detector Toolbox

For a more precise description of our toolbox, we introduce some terminology. Each detector works on a per-connection basis. A connection consists of two sequences of packets, one per direction: the originator sends packets $(p_1, p_2, \ldots, p_n)$ and the responder sends $(\overline{p_1}, \overline{p_2}, \ldots, \overline{p_m})$. As much of our discussion is symmetric in terms of directionality, here we consider only detection of originator-side activity. We indicate a packet's TCP flags by writing p^{flags}, where *flags* is a subset of $\{S, A, F, R\}$ corresponding to which of SYN, ACK, FIN, and RST are set. We use p^D to indicate a data packet (which will have ACK set, but not SYN, FIN, or RST). *seq*(p) is the sequence number of packet p; *ack*(p) the ACK number; *len*(p) the TCP payload length; and *time*(p) the packet's timestamp. When we compare sequence/ack numbers, we do so in accordance with TCP's sequence space (e.g., taking 32-bit wrap-arounds into account). $\tau(i)$ yields the largest index j so that *time*(p_i) $>$ *time*($\overline{p_j}$), pinpointing the most recent packet (relative to p_i) in the opposite direction. Finally, for easier notation we define a predicate earlier(p, *flags*, [*same-dir*$\|$*opp-dir*]), which is true if and only if there exists a packet earlier than p, sent by either the same endpoint (*same-dir*) or the opposite one (*opp-dir*), that has one of the specified flags set. If the direction is omitted, the predicate holds if such a packet has been seen in either direction. Using this terminology, Table 3 provides the precise definition for our detectors.

B Open-Source Injector Implementations

Although we did not have access to the tools/devices we detected in our datasets, there are open-source RST injectors available that we studied: two separate plug-ins for this task that come with the Snort NIDS [24]; the `rst` utility that comes with the Bro NIDS [19]; and *tcpkill*, a stand-alone tool for RST injection [25].

We find (as discussed below) that each tool crafts its RST packets somewhat differently. While the TCP standard mandates some packet header elements for injected RSTs (e.g., IP addresses and ports), other fields exhibit more freedom. In Section 4 we systematically discuss the range of choices available to an injector. Most injectors will send packets to both endhosts, reversing the SYN and ACK fields for the packets in the reverse direction.

Snort (as of version 2.8.1) has two plugins able to perform RST injection. The older plugin, *sp_respond* sends a single packet in each direction with a random IPID, a random TTL, and zero window size. The newer plugin, *sp_respond2* sends by default 3 RSTs to each endpoint. In each it sets the TTL to one of four values (depending on the triggering packet) and selects a random IPID. The first RST is initialized with the current SEQ number, with subsequent RSTs increasing the ACK number by half the TCP window size, but not incrementing the sequence number. Although *sp_respond2* has the same basic logic bug as the Bezeq injector of incrementing the ACK instead of the SEQ field, the different ACK increment and constant IPID for the Bezeq injector suggest that these are independent implementations.

The Bro NIDS comes with an external tool, *rst*, which takes the connection's 4-tuple as well as the most recently observed sequence numbers as arguments. The injected RSTs have a TTL of 255, IPID and window size of 0, and the SEQ and ACK value from the arguments. The tool generates a controllable number of RSTs in each direction; if sending more than one, then it also inserts fake data packets with rising sequence numbers in between to attempt to advance the sequence point if the first RST is ignored, with each data packet is followed by an in-sequence RST.

tcpkill ships as part of the *dsniff* toolbox and is the only injector that operates in a single direction. It monitors a network link via *libpcap* and selects a subset of TCP packets as specified by a user-supplied BPF expression. For each (non-control) packet, *tcpkill* sends (by default) three RSTs back to the packet's source address. When building the RSTs, it sets the TTL to 64, picks a random IPID, keeps the packet's window size, and sets the sequence number to the ACK number plus i times the window size, with $i = 0..2$ according to the number of the RST sent. It sets the RST's ACK number to zero.

Although all these injectors have potential fingerprints, we did not notice any of them being used in a significant amount.

C Real-World IPIDs and TTLs

In initial experiments aimed at understanding which of a RST's features an injection detector can rely on, we examined IPID and TTL values in depth before concluding that they did not provide suitable criteria for detecting injected RSTs.

As these fields can in principle be freely chosen by an injector (see Section 4), we thought that at least a subset of forged RSTs would be detectable by observing inconsistent choices within individual flows. However, as is often the case due to network traffic's variability, we found that these values are highly volatile even within normal network traffic. To demonstrate this, we examined one week of our research institute's border traffic, starting on April 18, 2008. The dataset included 4,033,204 flows, 25.0% of which had more than 10 packets from either the source or the destination.

We started by testing whether the results on the prevalence of RST traffic from [1] held. Of all flows, about 5% were terminated with an originator-side RST and 0.6% with a responder-side RST. While lower than the 15% figure in the original study, the general observation still holds: a significant portion of connections are terminated via RSTs.

In general, we found that the TTLs of the RST packets varied markedly from the previous data packet. Examining only RST-terminated flows, for about 7% of those terminated by the originator the RST packet's TTL differed; this rose to 28% for responder-terminated flows. We might expect such TTL differences to be minor, but in fact the volatility was often very high, with TTL changes clustering around 64, 96, 128, and 192, with a significant number of seemingly arbitrary differences.[3]

We also confirmed that affected flows were not particularly unusual. We randomly selected 200 flows where the RST packets had a differing TTL, 20 flows where the client was volatile and 20 where the server was volatile, in each of 5 TTL ranges. Of these, only two flows appeared to be unusual (these flows triggered our detector).

Thus, we conclude that the recommendation in [5] to ignore RST packets with unusual TTLs will suffer from significant false positives.

We also examined the IPID volatility on these reset connections. For originator-terminated connections, 36% used an increment consistent with the current flow; 34% were four times the normal increment; a bit under 1% had a RST IPID of 0; another 1% used the same IPID as the previous packet; a bit over 1% used twice the normal increment; 0.5% used three times the current increment; and 27% had no apparent relation. We found a similar distribution for responder-terminated connections. Thus, although we use both TTL and IPID to fingerprint injectors (Section 7), we do not find these to be effective distinguishers of injected RST packets.

[3] We also observed a similar level of TTL volatility between SYNs and data packets, as well as between data packets and FINs.

Name	*Description*	*Definition*
RST_SEQ_DATA	Outdated RST following data.	(p_i^D, p_{i+1}^R) , where $\mathit{seq}(p_{i+1}) < \mathit{seq}(p_i) + \mathit{len}(p_{i+1}),$ and $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_1,$ and $\neg$ earlier$(p_i, F \mid R)$
DATA_SEQ_RST	Data following a RST.	(p_i^R, p_{i+1}^D) , where $\mathit{seq}(p_{i+1}) + \mathit{len}(p_{i+1}) > \min_{j \leq i} \mathit{seq}(p_j^R),$ and $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_1$
RST_SEQ_CHANGE	Multiple RSTs with increasing seq.	(p_i^R, p_{i+1}^R) , where $\mathit{seq}(p_{i+1}) > \mathit{seq}(p_i) + 2,$ and $\mathit{seq}(p_{i+1}) > \max_{j<i} \mathit{seq}(p_j),$ and $\mathit{seq}(p_{i+1}) > \max_{j \leq \tau(i)} \mathit{ack}(\overline{p_j}) + 2,$ and $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_1,$ and $\neg$ earlier$(p_i, F),$ and $\neg$ earlier$(p_i, R, \mathit{opp\text{-}dir})$
RST_ACK_CHANGE	Multiple RSTs with increasing ack.	(p_i^R, p_{i+1}^R) , where $\mathit{ack}(p_{i+1}) \notin \{\mathit{ack}(p_i), \mathit{seq}(p_i), 0\},$ and $\mathit{ack}(p_{i+1}) > \max_{j \leq \tau(i)} \mathit{seq}(\overline{p_j}) + 2,$ and $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_1,$ and $\neg$ earlier$(p_i, F),$ and $\neg$ earlier$(p_i, R, \mathit{opp\text{-}dir})$
SYN_RST	RST after SYN.	(p_i^S, p_{i+1}^R) , where $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_2,$ and $\neg$ earlier$(p_i, \mathit{any}, \mathit{opp\text{-}dir})$
SYN_ACK_RST	RST after SYN/ACK.	(p_i^{SA}, p_{i+1}^R) , where $\mathit{time}(p_{i+1}) - \mathit{time}(p_i) < T_2,$ and $\neg$ earlier$(p_i, \mathit{any}, \mathit{opp\text{-}dir})$

Table 3. Detector Toolbox. See Appendix A for terminology and Section 5 for the rationale behind choosing $T_1 = 2$ sec and $T_2 = 0.1$ sec.