fling: A Flexible Ping for Middlebox Measurements

Runa Barik, Michael Welzl
Department of Informatics,
University of Oslo, Oslo, Norway
{runabk,michawe}@ifi.uio.no

Ahmed Elmokashfi
Simula Research Laboratory, Norway
ahmed@simula.no

Stein Gjessing, Safiqul Islam
Department of Informatics,
University of Oslo, Oslo, Norway
{steing,safiquli}@ifi.uio.no

***Abstract*—Middleboxes in private networks have been known to change packets in many ways, making it hard to design protocol extensions that work for the large majority of Internet users. Addressing the need to know what such middleboxes do, we introduce a tool called *fling* ("flexible ping"). *fling* can carry out (almost) any kind of protocol dialogue between a server and a client based on a simple specification in a json and a pcap file, and identify what middleboxes do to the packets of the dialogue. This fills a gap in the state of the art, where other tools that control both ends of a path are either limited in some form or have to be updated for every new test. We present results from small-scale tests that prove the flexibility of *fling*, which is a prerequisite for our next step: development of a large-scale measurement platform.**

***Index Terms*—Middlebox, Measurement, fling**

I. Introduction

When developing protocol extensions in the IETF, it is often important to know what will happen to particular types of packets along a path ("If we add an option to this packet, is it more likely to be dropped? Will the option often be removed?"). Such considerations have played a major role in the design of Multipath TCP (MPTCP) [40]. Middleboxes have also been shown to harm network measurements per se [13]. Our previous work [11] investigated the effects of middleboxes on certain fields of the IP header, and found that nonzero DSCP values may provoke consistent packet loss. This has affected the IETF rtcweb standard[1]. While this small measurement study already used a preliminary prototype of *fling*, this is the first time that we fully describe the tool itself.[2]

During the past decade, deployment of large scale measurement infrastructures has become popular both to inform policy makers and help users gain insights into their network performance (e.g., M-Lab, RIPE Atlas, BISmark, SamKnows, etc.). A good overview of existing measurement platforms is given in [9]. Most of them use customized variants of ping, traceroute, ntp, netstat, iperf, etc. to measure various network aspects, but they usually do not focus on analyzing the influence of middleboxes on traffic. Some can detect middleboxes to some extent: for example, Netalyzr [29] sends TCP and UDP packets to test port reachability and test for proxies, uses an ESP header over UDP packets to detect IPSec NATs, performs Path MTU Discovery and examines the effect of IP fragmentation.

A number of tools were designed to specifically measure the impact of middleboxes. For instance, TCPExposure [25] tests certain TCP options between clients and a dedicated server, and tracebox [17] combines middlebox testing with traceroute to deduce packet changes from the payload of ICMP Echo Reply messages. Generally, doing tests that require more than normal user privileges from people's homes, which is where most problematic middleboxes are expected, is a difficult matter. It has been achieved via payment, e.g. to ship dedicated hardware [5], [44] or pay users to run a tool [32] (which creates a natural scalability limit), or by limiting the campaign to one-time tests, where testers are personally asked to run a tool that was created for a particular test. Such a one-time measurement campaign was done with TCPExposure [25], and the difficulty of repeating experiments is one of the lessons learnt according to the authors [24].

We introduce *fling*—our attempt to learn from these past success stories and combine them in a way that makes it easy to repeat two-sided middlebox measurements, even when they require administrator privileges. *fling* is an end-to-end active measurement tool that allows testing whether an arbitrary sequence of packets can be exchanged between a *fling* client and a *fling* server. These packets are defined in a PCAP file, which is accompanied with a JSON file that describes a dialogue. Tests are uploaded to the server and pulled by *fling* clients whenever they run, such that clients do not need to be updated whenever new tests are defined.

fling resembles ping in that it sends a number of packets from a *fling* client to a *fling* server and expects a few packets in return. Different from ping, these packets are not all equal – and *fling* executes a dialogue that is defined per test and can be much longer and more complex than ping's exchange of two packets. The security implications of ping are well known because ping is simple and well understood. We expect the same to be the case for *fling*—at the same time, we tried to make it as flexible as possible. Naturally, there are limitations to this flexibility (100% flexibility can only be achieved by installing new code for each measurement, which we wanted to avoid). Like ping, *fling* is not intended and cannot be (reasonably) used for bandwidth measurements; it is meant for sending and receiving a handful of packets and seeing what happened to them.

[1] https://www.ietf.org/mail-archive/web/tsvwg/current/msg14431.html

[2] We also presented an earlier prototype at the IMC 2016 Works-in-Progress Session.

978-0-9883-0453-6/17 $31.00 © 2017 ITC
DOI 10.1109/ITC.2017.23

TCPExposure is very similar to *fling*: it supports a client-server dialogue of any type of packets and observes what happens to them along the path. However, with TCPExposure, for every new test, the code would have to be updated, and users would have to be asked to install it and run an experiment. We wanted to develop a static tool that would yield this flexibility without requiring to get in touch with users and ask them to install new code.

Netalyzr is attractive for users to run as it lets them learn about their own network connectivity. It managed to attract a large user base. We tried to learn from that lesson by offering the same type of feedback to users that download and run *fling* on their home machines; in our case, the effort to use it is higher (it is not embedded in the browser, because we need to use raw sockets) but the range of possible outputs is wider (because we use raw sockets). Also similar to Netalyzr, it is possible to update tests by changing a dedicated server only. Like tracebox, *fling* also identifies *where* on the path a packet change or drop occurred.

After an overview of related work in the next section, section III will introduce the design of *fling*, including a discussion of its inevitable limitations. We have put these limitations to the test with a small measurement study, which we report about in Section IV. Section V concludes the paper and discusses our next steps towards the development of a permanent *fling* measurement platform.

II. Related Work

The increasing popularity of middleboxes has motivated several efforts to characterize their deployment and assess their impact on data plane performance. Medina et al. [34], [35] actively probed a set of web servers using TBIT [39] to assess the interaction between middleboxes and transport protocols. Honda et al. [25] developed TCPExposure to test whether TCP options are supported. TraceBox [17] improved over TCPExposure by proposing a Traceroute-like approach to pinpoint routers that alter or discard TCP options. Cravan et al. [16] proposed TCP HICCUPS, a tool that reveals TCP header manipulation to both ends of a TCP connection. *PATHspider* [31] is a recent tool that allows for A/B testing of a baseline configuration against an experimental configuration.

Table I provides an overview of the measurement tools mentioned above, and shows how they compare to *fling*. The first two columns illustrate a limitation of prior work that *fling* addresses: while almost all of the tools use raw sockets, potentially allowing them to transfer *any* type of Internet packet, experiments so far have mostly been limited to TCP over IP: the TCP header's source port, initial sequence number, window and option fields were the focus of [16], [17], [25], the IP header's DSCP, ECN, flags, source address and option fields were considered in [11], [21], [34], [35], [37]–[39], [45], [46], while, to the best of our knowledge, the only recent middlebox measurement studies considering protocols such as UDP, SCTP and DCCP are [22], [33] and [19].

Other papers focused on investigating specific types of middleboxes such as web proxies [47], transparent HTTP proxies in cellular networks [48], firewalls and NATs policies in cellular networks [46], and carrier grade NATs [37]. Trammell et al. [45] have proposed correlating measurements from diverse vantage points to build a map of middlebox-induced path impairments in the Internet.

fling bears some resemblance to pcap replaying tools such as tcpreplay [6] and its variants; it differs in that *fling* is two-sided, describing a complete dialogue, with timeouts, behaviour that is triggered by the reception of packets, etc. We will now turn to a full description of *fling*'s design.

III. *fling* Design

Ping sends a number of specific packets (typically ICMP Echo Request) to a host and expects to get the same number of reply packets (typically ICMP Echo Reply). In essence, this is also what *fling* does, but it adds flexibility: *any* packet can be used instead of ICMP Echo Requests, and the dialogue can take any form, involving multiple packets (e.g. in case of TCP, using only single packets would limit *fling* tests to SYN-SYN/ACK tests).

```
{"name":"TCP SYN/ACK test",
"__index":{"0":"TCP SYN",
                "1":"TCP SYN/ACK"},
"packet_Info":[{
                "name":"TCP SYN",
                "swap":[0,2,2],
                "ChksumType":"adler-32",
                "ChksumPos":[16,0],
                "ChksumLen":[2,0],
                "ChksumPseudoHDR":true
                },{
                "name":"TCP SYN/ACK",
                "swap":[0,2,2],
                "ChksumType":"adler-32",
                "ChksumPos":[16,0],
                "ChksumLen":[2,0],
                "ChksumPseudoHDR":true}],
"client":{
        "state_sequence":["S1"],
        "states":[{
                "state":"S1",
                "send":["TCP SYN"],
                "recv":["TCP SYN/ACK"],
                "delaySend":[0],
                "timeout":[2000]
                }]},
"server":{
        "state_sequence":["S1","S2"],
        "states":[{
                "state":"S1",
                "recv":["TCP SYN"],
                "timeout":[2000]
                },{
                "state":"S2",
                "send":["TCP SYN/ACK"],
                "delaySend":[0],
                "timeout":[2000]
}]}}
```

Figure 1: json file for a simple TCP SYN-SYN/ACK dialogue test

A *fling* client is a static piece of software; it begins a test by pulling a test description (a pcap file containing the test packets and a json file describing the test) from the server, which it then executes. A test description specifies the packet types, some information about header fields, and the sending/receiving sequences of the dialogue. It never contains

Tool	Raw sockets	Test protocols other than TCP	Test update: need to change	Fully controlled client-server dialogue	Detect TCP connection splitters	tracebox-like location detection
fling	✓	✓	Server	✓	✓	✓
Netalyzr	✗	✓*	Server	✓	✓**	✓‡
TCPExposure	✓	✗	Both	✓	✓	✗
HICCUPS	✓	✗	Both	✓	✓	✗
Tracebox	✓	✓	Client	✗	✓	✓‡
PATHspider	✓	✓	Client	✗	✓	✓‡
TBit	✓	✗	Client	✗	✗	✗

Table I: Comparison of related tools. *ICMP,UDP; ‡One-sided only; ** only HTTP proxies.

addresses: a *fling* client always only talks to a specified (supported as a command-line option) *fling* server. This allows to fully control the dialogue and collect measurement results at the server for research use; it also serves as a security measure, by ensuring that attackers cannot design tests that would turn *fling* clients into sources of traffic towards some other hosts in the network. To avoid getting in the way of normal Internet usage of *fling* users, the total maximum number of packets transmitted by *fling* is also statically configurable by the client.

An example json file is shown in figure 1; both the client and server execute it after an HTTPS handshake (this is the control channel, which we will explain in the next section). Every *fling* test begins with at least one packet from the client. In our example, the client sends a SYN packet and then waits for up to 2000 ms to receive a SYN/ACK packet which it stores upon reception. Starting from the point where it answers the HTTP request (see Fig. 2), the server waits for up to 2000 ms to receive a SYN packet which it stores upon reception (state "S1"). Then, either upon timer expiry or immediately after receiving the packet, it enters state "S2" and sends a SYN/ACK in response.

In a *fling* experiment description, every state contains a timeout, which gives a limit for the duration of the state. Packets are logged until the state is over. The entries "send" or "recv" are used to transmit packets or expect the reception of packets, respectively. Receiving packets in accordance with "recv" terminates a state before the timeout. For each state, multiple packets can be specified to be sent or received, and each transmission can be accompanied by a "delaySend" value: the time that *fling* waits before sending a packet. When a state contains "send", the state's timeout begins after the last packet was sent.

Figure 1 also shows the common header of a *fling* experiment description. It contains the name of the experiment as well as an index entry that maps the pcap file packet numbers to names in the description text (in this example, the pcap file contains a SYN packet, followed by a SYN/ACK packet). The "packet_info" statement contains information about port swapping and checksums; we will explain this later.

A. The Fling Control Channel

fling is not meant to be shipped with a specific set of tests; rather, it obtains tests from a *fling* server. This query is made using HTTPS, initiated by the client (because we assume that many *fling* clients would operate behind a NAT). We call this HTTPS communication the Fling Control Channel (FCC) because it is used to transmit more control information—among them, a nonce that the client inserts into *fling* packets, at a position that can be defined per packet as part of the test description.

The nonce allows the server to identify *fling* packets. Because the idea of *fling* is to allow exchanging *any* packet type, we cannot rely on common methods to determine where a packet comes from. Consider, for example, two clients behind the same NAT, doing an ICMP test: because ICMP has no port numbers, we need our own identifier to tell these clients apart.

The FCC also lets us transport the client's received *fling* packets back to the server as HTTPS payload for further analysis, and it is used to control experiment initiation and termination. As shown in the Fig. 2, a session begins with the client sending an HTTPS GET request containing an empty "Hello" message. The server answers with a test number, the pcap and json files, a *salt* value for nonce generation and a dictionary that contains a random number for each *fling* packet. It also starts the first timer to wait for incoming *fling* packets. When the client gets the HTTP response, it starts the test by transmitting the first *fling* packet. When the test is finished, the client sends its results (all logged packets that it received) as HTTPS payload to the server, and the server responds with some further data about the test that is useful for the client to give feedback to the user.

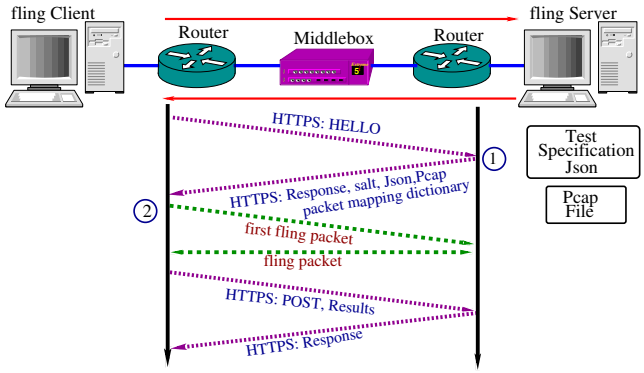

Figure 2: *fling* client and server interaction. The server starts the first timer at (1). At (2), the client begins by sending the first *fling* packet and starting its first timer.

B. Security and NATs

fling's nonce lets the server associate *fling* packets with the preceding HTTPS handshake. It also prevents a reflector attack, where an attacker would alter a *fling* client to provoke the server to send packets back to a spoofed source address. The nonce is the concatenation of the 8-bit *salt* value per experiment and a random number generated for each packet.

In the interest of flexibility, we allow freedom to choose the length of the nonce and to specify where in the packet it should be written. For the position of the nonce, an experiment designer could use any parts of a header that are likely to be immutable or—for data packets—use the payload. To handle cases where the nonce is in the payload and middleboxes insert header fields (e.g. options), the nonce offset can also be provided from the end of the packet (decided by optionally including a "fromBack" attribute in the json file). The default nonce length is 16 bit and the default position is the mostly unused Identification field in the IP header.

We chose a field of the IP header as a default position because we cannot make a default assumption about packet headers following IP; for *fling*, *anything* can be there. This also explains the need to specify details about a checksum ("packet_Info" in figure 1): if the nonce is written into a transport header or payload, this header's checksum will need to be recalculated. The value cannot be known beforehand because the nonce is calculated at run time.

fling allows a minimum nonce length of 8 bit (0 bit for the random numbers; such experiments can only contain one packet from each side). Because the nonce can be so small, the server also limits its responses to tests from the same IP address that was used on the HTTPS connection. Using a short nonce increases the chance for an attacker that is behind the same NAT as a regular *fling* client (or changes its source address to match an ongoing *fling* test) to inject wrong packets that the server would be forced to accept.

In a truly end-to-end Internet, the transport header should not matter to routers and *fling* should be able to do whatever it wants on top of IP. This notion is, however, already broken by NATs, which, in practice, often carry out NAPT (Network Address and Port Translation) [42]. When a client sends packets of a known transport protocol (e.g. TCP or UDP) from behind a NAT, the server cannot just apply a statically-defined transport header, but it needs to swap the NAT-written port numbers. Whether this functionality is desired or not depends on the packet format (e.g., ICMP does not have ports). Therefore, if "packet_Info" in the test description contains "swap: [*loc1*,*loc2*,*len*]" where *loc1* is the location of source port, *loc2* is the location of the destination port and *len* is the length of ports in bytes, the server sets the ports accordingly instead of taking them from the pcap file. This is another reason to carry out a checksum calculation. The attributes "swap" and "ChksumPos" are specified relative to the front of the transport header.

"swap" is in fact shorthand for two "copy" operations that copy fields from previously arrived packets. An example from our measurement campaign that highlights the need of such a generic "copy" operation is discussed in section IV-A.

C. Narrowing Down the Root Cause of Packet Drops

Given that *fling* is about reachability and tries to detect what middleboxes do to packets, we should be able to detect the rough location of middleboxes that change or drop packets. Moreover, if a *fling* packet is dropped on a path, we need to ensure that it happened due to the middlebox's behavior, not due to congestion. Hence, we rely on using additional packets that we call *anchor packets*. Our anchor packets are either of type ICMP Echo Request or TCP SYN, and we answer them with ICMP Echo Reply or TCP SYN/ACK packets, respectively. Since NAT boxes use the *id* and *seq* fields of the ICMP header to map ICMP reply to request packets, we also maintain the correct values for the *id* and *seq* fields of the request and reply packets [41]. Since both the client and server need to be able to associate anchor packets to their corresponding *fling* packets, we store the last 16 bits of the nonce value in the IP header's 16-bit Identification field of the anchor packet (if the nonce is smaller, the remaining bits are set to 0). Note that any kind of anchor packet can be defined as part of the test description itself, making the test description slightly more complex but allowing full flexibility for the placement of the nonce.

Anchor packets have strictly two purposes: 1) detect congestion, 2) trigger a tracebox-like test. They are *not* meant as a replacement for A/B-measurements, where type A packets would differ from type B packets in a particular way and it could be established that the difference between A and B caused an effect. Such measurements can easily be designed with *fling* by including both type A and B packets in a test description. A *fling* test could also easily define its own special packets to be sent in conjunction with all other measurement packets ("send" in the json syntax operates on arrays, making it easy to send multiple packets), as a way of implementing customized anchor packets. We hard-coded ICMP Echo Request/Reply and TCP SYN-SYN/ACK purely for convenience.

We send an anchor packet immediately ahead of every outgoing *fling* packet. The idea is that repeatedly losing both packets together gives an indication that the packets were dropped due to congestion (refer to Table II). More importantly, if the anchor packet repeatedly arrives but the *fling* packet does not, this is a strong indication of a middlebox-induced packet drop. Thus, if a *fling* packet is dropped we repeat the test up to three times. This number is configurable.

Fig. 3 shows how the client detects a packet drop using anchor packets. After knowing that a *fling* packet was dropped, we try to find the location of the drop by repeatedly sending the same *fling* packet with a growing TTL starting from 1. As soon as no ICMP TTL exceeded error packet arrives, we know the location of the packet drop. Because we send the results of these tests back to the server in the final HTTPS exchange, the server can also identify changes that may have happened to the *fling* packet given that responding routers are RFC1812-compliant [10]. This approach is similar to *tracebox* [17].

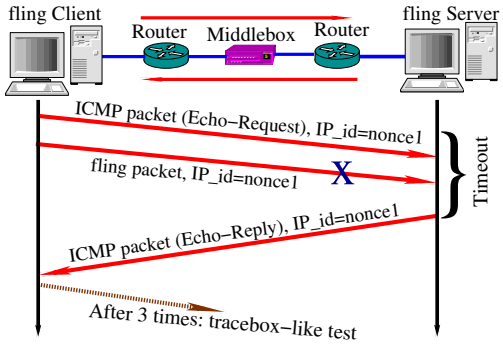

Figure 3: Detection of *fling* packet drop and middlebox location

***fling* packet**	***anchor* packet**	**Interpretation**
PASSED	PASSED	SUCCESS
PASSED	DROPPED	SUCCESS
DROPPED	PASSED	Repeat 3×. Then, assume: MIDDLEBOX DROP; start tracebox-like test
DROPPED	DROPPED	Repeat 3×. Then, assume: CONGESTION; start tracebox-like test

Table II: Interpretation of arriving or dropped *fling* / *anchor* packets. CONGESTION: in one type of *fling* test, *anchor* packet passes and in another, it is dropped. SUCCESS: all *fling* packets passed; if packets have changed in transit, start tracebox-like test.

IV. Evaluation

We have prototyped *fling* in Python, based packet capturing and manipulation on Scapy [7], and prepared tests for 36 distinct protocols and protocol options. By preparing a test, we mean that we have generated all needed packet traces to test a protocol or an option. We then asked friends and colleagues to run *fling*. In total, 34 users ran *fling*, and each test was run against three servers simultaneously to avoid any server-related bias, giving us a total of 3384 tests. One of the three servers was hosted on Amazon EC2 cloud and the other two were hosted in Norway, where they were connected to the Internet via two different ISPs. The tests originated from nine countries with over half of them coming from Norway and Austria. Further, about two thirds of users stated that they ran the tests from their homes.

Next, we present our tests and the trial run results. Note that the goal of these runs is not to make a general statements about the support of specific protocols in the Internet, but rather to examine *fling*'s flexibility. We, however, plan to conduct large scale measurement campaigns in the future.

A. Test Design

To evaluate the flexibility of *fling*, we first analyzed middlebox measurements from previous work to try to understand whether we could replicate them. These measurements include:

- The ECN tests from TBIT [34], [35], [38], [39] and [30].
- The TBIT TCP options tests, and the IP options tests from TBIT and [21].
- The TCPExposure [25] tests: using *MP_CAPABLE*, *MP_DATA*, *MP_ACK*, the Timestamps option, and a TCP handhake followed by data and ACK packets (containing SACK) to check for sequence number changes.
- The HICCUPS [16] tests, where special numbers are inserted in the sequence number, IP Identification and receive window fields to convey integrity information.

We find that *fling* can replicate almost all of these tests, with only very minor limitations: because it relies on the nonce, *fling* cannot detect TCP splitters in the same way TCPExposure does; it does however detect them during its tracebox-like test phase. In some cases, a dynamic decision taken in a test must be replaced with separate static tests. For example, two static tests for SYN-SYN/ACK handshaking with or without ECN set-up are needed to replicate the TBIT ECN test; similarly, two tests are needed to conditionally answer a packet containing the *MP_DATA* option with *MP_ACK* or not, depending on whether *MP_DATA* passed through the path (this is done by TCPExposure). Finally, in HICCUPS [16], the values of the altered fields in the SYN/ACK packet are the result of a computation on the received SYN packet. Because it does not allow to define arbitrary calculations on header fields, *fling* cannot truly replicate the behavior of HICCUPS, but it can detect all the header changes that HICCUPS also detects.

Next, we decided to run a variety of protocols over IP, and do tests with changes applied to the IP header (e.g. testing options or unknown protocol numbers) and the TCP header (e.g. testing options or using a wrong value in the Data Offset field). The complete set of tests that we carried out is listed in Table III; these were pure *fling* tests, i.e. packets of the described type were transmitted between the clients and the server to see what would happen to them along the path. This table also shows how many tests succeeded. We only decided that a test failed when all three repetitions failed; if, in these three failures, anchoring ICMP messages were dropped but ICMP messages passed from the same client at least once in another test, we decided that this failure could have been due to congestion and removed the test from our set. There was only one such case.

Again, the static nature of *fling* highlighted a handful of limitations: RSVP requires to put IP addresses and port numbers in the RSVP header. At first, we could not do this; this prompted us to devise the aforementioned generic "copy" operation. Some protocols, however, require calculations, which *fling* cannot do: a Quick-Start [20] recipient (server) should generate entries in its response by computing the TTL difference as (IP_TTL–QS_TTL) mod 256, and retrieving the allowed rate request from the received IP option. Similarly, the AH *Integrity Check Value (ICV)* in the OSPF/AH test is wrong because *fling* changes the underlying IPv4 header but does not recalculate this value (section 3.3.3 of [28]).

Complete flexibility can only be attained by installing new code for each test, or allowing to specify an actual protocol

(arbitrary operations on header fields based on prior received headers and local state). With *fling*, we intend to strike a balance in trying to be simple yet flexible. We conclude from our test design study that, despite being unable to do absolutely all tests, the number and diversity of tests that *fling* can do is indeed large: it was able to replicate the large majority of tests from existing work and allowed us to carry out truly "crazy" tests involving e.g. changes to the IP header that follow an obsolete specification, wrong field combinations in TCP, or transmit OSPF, HIP and DCCP packets end-to-end.

Next, we briefly evaluate the results of our tests. Our test set is small: at this stage, our intention is to test-drive *fling* before we roll it out on a larger scale. Thus, we do not try to derive broad statements about the Internet from our measurements, but we want to ensure that such statements *could* be derived in a larger-scale study.

B. Results

Table III shows the tests that we carried out. Unless otherwise noted, all protocols were used directly over IP, and Scapy defaults apply to header fields. IP and TCP header tests used Scapy-generated packets: a TCP SYN from client to server (src port 48001, dst port 443), and a TCP SYN/ACK in response. The "Success ratio" column shows the fraction of successful runs (i.e., all *fling* packets reached the other side within 3 trials) for each test.

As expected, no protocol has a 100% success rate except for UDP. Broadly speaking, our tests can be classified into four categories: IP header changes, IP protocols, TCP options, and transport protocols other than TCP. Packets with IP header changes were among the least likely to pass end-to-end, confirming earlier observations about IP options [21]. While an unknown TCP option worked in 50% of the cases, a wrong Data Offset value worked in only 5%. This indicates that the success of an unknown option does not mean that middleboxes do not look inside the TCP header; they *do* look, but they often allow unknown options to pass.

SCTP appears to enjoy a decent support with 2/3 of the tests succeeding; this Internet test confirms the local testbed result in [22]. We inspected all failed tests and found that in over 95% of cases a test failed to all three servers, which indicates that the packet drop happened close to the client. IP option tests succeeded only from one client to one server, which both were from the same autonomous system, indicating different blocking policies for intra and inter domain traffic.

As mentioned above, once a server or a client detects that a test has failed, it attempts to determine the packet drop location by repeatedly sending the dropped packet with a growing TTL starting from one. Figure 4 shows the packet drop location in terms of the number of hops. These plots show that most of the blocking either happens at the client's immediate gateway or two hops away. Further, there is a significant fraction of blocking that happens several hops away from the client on the forward path. This, however, is not the case on the reverse path. Since the tests are always client-initiated, the above observations hint that whenever a test succeeds on the forward path, it is likely to succeed on the reverse path unless the client's immediate gateway blocks incoming packets. Going forward, we plan to take a closer look at what causes the blocking several hops away from the client and add functionalities to *fling* that—depending on user consent—fingerprint home gateways.

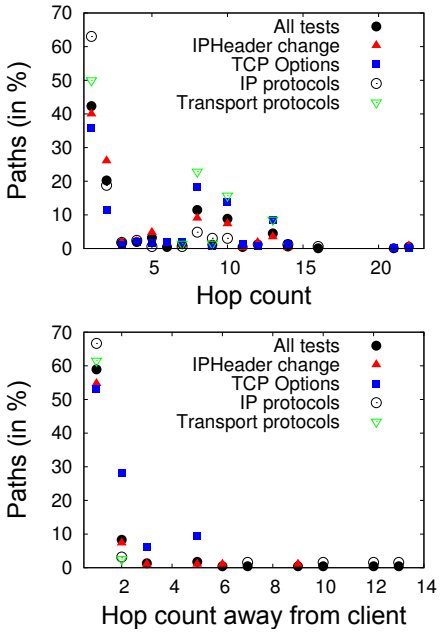

Figure 4: Packet drop location: forward path (top), and reverse path (bottom). The reverse path was determined from the server, but it is shown as the number of hops from the client.

Since *fling* servers and clients capture all exchanged packets, we can also investigate whether packet headers and options were modified. Inspecting packets from successful TCP tests, we find several cases of altered TCP MSS options and option removal. For example, the reserved TCP option 100 was removed in 14 experiments and the MPTCP option was removed in three experiments.

V. Conclusion and Next Steps

Our prototype tests have shown that *fling* is indeed very flexible, allowing for a wide range of middlebox tests. Our tool produces data that lets us better understand if, how, and where middleboxes influence packets of a certain type as a result of the dialogue that they witness.

A. Towards a permanent fling platform

So far, we have described and used *fling* as a tool that a user downloads and runs once, yielding information about the network path between the user's host and our server.[3] For the "test-drive" measurement campaign described in this paper, we had to ask users to run the tool, similar to how TCPExposure tests were carried out [25]. The true goal of

[3]This "one-time" *fling* version is GPL licensed and available from the main *fling* webpage.

Test description	Success ratio
Routing and Addressing – **Initiation of a HIP session:** *I1* (a HIP Initiator Packet) is sent to the *fling* server, and *R1* (HIP Responder Packet) is sent to the *fling* client. On success, the client sends *I2* and in response, it receives *R2* (Section 5.3 of [36]). We ran the HIP package from [1] to collect HIP packets and observed HIP packets over UDP, which we extracted to run HIP over IP.	0.51
QoS – **Initiating an RSVP reservation:** The *fling* client sends a *Path* message to the server, and the server responds with a *Resv* message. We used two packets from the pcap file from [4].	0.0
Tunneling – **ICMPv6/IPv6:** ICMPv6 *Echo request* with non-existing IPv6 address pair is sent to the *fling* server which responds with *Echo reply*. We used 2 packets from the pcap file from [2].	0.5
ICMPv4/IP/GRE: We set *checksum present* = 0 in the GRE header. An ICMPv4 *Echo request* packet is put in an IP/GRE tunnel and sent to the server. The server responds with an ICMPv4 *Echo reply* packet over IP/GRE. We used 2 packets from the pcap file from [2].	0.66
Security – **ICMPv4/IP/ESP (Tunnel Mode):** The client sends an ESP packet containing an ICMPv4/IP *Echo request* packet as encrypted payload. The server responds with an ESP packet containing an ICMPv4/IP *Echo reply* as encrypted payload. We used two packets from the pcap file from [8], where it is explained how to decrypt the data stream. We did this to identify the packets.	0.649
AH (Transport Mode): The *fling* client sends an AH packet to the *fling* server, and the *fling* server answers with an AH packet. We used two packets from the pcap file from [2].	0.585
Transport – **SCTP association establishment:** The client sends an SCTP *INIT* packet with src port 48001 to the server on port 443, and the server answers with an *INIT_ACK* packet. Then, the client responds with *COOKIE_ECHO* and the server responds with *COOKIE_ACK*.	0.66
UDP: The client sends a UDP packet with src. port 48001 to the server on port 443, containing 4 bytes of data; the server responds with a similar UDP packet (but flipped ports).	1.0
UDP-Lite: The client sends a UDP-Lite packet with source port 32768 and destination port 1234, with *checksum coverage* = 0, containing 12 bytes of data; the server responds with a similar UDP-Lite packet (but flipped ports). We used two packets from the pcap file from [4].	0.511
Complete DCCP session: The client sends a DCCP *Request* packet with src port 32772 to port 5001 and the server answers with a *Response* packet. The client answers with an *Ack* and a *DataAck* packet containing 256 bytes of data. The server responds with an *Ack*. Finally, the client transmits a *Close* packet and the server responds with a *Reset* packet. We used the pcap file from [4], shortened the data transfer and adjusted the sequence numbers of the closing packets.	0.511
IPv4 options – **Quick-Start (QS) (QS request [20]):** The client sends a packet containing a QS *Request* option with IP TTL 64, *QS TTL* 90, IP option number 25, option length 8, *Function Value* 0, *Rate Request* 5 (1.28 Mbit/s), *QS Nonce* 2, and the *reserved* field (2 bits) set to zero. The server responds with a QS *Response* option as a TCP option in the SYN/ACK packet with *Rate Request* 5, *TTL Diff* 5, *QS Nonce* 2, and all *reserved* fields = 0.	0.053
Router Alert [26]: The same option was used on both packets, with *value* set to zero.	0.053
Security and Extended Security (historic) [27]: We made two tests: one for *Basic Security* where *Classification Level* is set to *Secret* and *SCI* and *NSA* bits set for *Protection Authority Flags*; and another for *Extended Security* (zero is set for security info and its code) along with *Basic Security* options. Both SYN and SYN/ACK packets carry these options.	0.011
Other IPv4 header changes – **DiffServ CodePoint (DSCP):** We tested some DSCP values from [18] which proposes to opportunistically set them for WebRTC: CS1 (8), AF42, EF PHB (46).	0.75
"Evil" bit [12]: The client and server exchange SYN-SYN/ACK packets with this bit set.	0.745
Unknown Protocol numbers (143, 200, 252, 253, 255): The client sends a TCP SYN and the server responds with a TCP SYN/ACK, but both use the same unknown protocol number.	0.51
TBIT test 2, IP Option X (option number 31): We carried this test out statically, i.e. without retrying three times in case of failure.	0.053
IP options tests from [21]: The client sends a TCP SYN with an IP option to the server on port 80 and the returns it on a TCP SYN/ACK. We generated the packets with *Scapy*.	0.06
TCP header changes – **TCP Fast Open (TFO) [14]:** We downloaded the pcap file from [3], removed two unnecessary packets (an intermediate GET and a final ACK) and edited the packets to test option kinds 34 (newly allocated) and 254 (experimental). For 254: The client sends a SYN packet with *magic number* 63881 and requests a *cookie* (kind 34 does not use a magic number, but requires padding). The server responds with a SYN/ACK packet containing the same *magic number* and a *cookie*. The client then sends an HTTP request inside a SYN/Fast Open packet with the same *magic number* and *cookie* value. The server responds with a SYN/ACK.	0.628
TCP mood (Test for TCP Happy packet (April 1 RFC [23])): The *fling* client sends a SYN packet with an option of kind 25 and value 14889 (":)" in ascii) for *Happy* mood, and the server sends a SYN/ACK with a happy mood too.	0.755
TCP NoP (Test for NoP option / wrong Data Offset): This tests what happens if the Data Offset value in the TCP header is wrong, both for a SYN from the client and the corresponding SYN/ACK from the server. The IP *Length* field says that the packet is 42 bytes long and the TCP Data Offset value is 6, meaning 24 bytes. In reality, there are 20 bytes of regular TCP header, 2 bytes of NoP options, and the last 2 bytes do not exist in this packet.	0.053
TCP unknown option (Test for a large unknown TCP option using reserved option number 100): The client sends a SYN packet with option kind 100, a correct length field and 40 bytes option content; the server sends a SYN/ACK with the same option.	0.5
TBIT-based ECN test: The client sends a SYN packet with set ECN_ECHO and CWR flags to a web server on port 80, and the server sets the the ECN_ECHO flag in the SYN/ACK response. This handshake is (in TBIT, only in case of successful ECN negotiation) followed by an HTTP request with ECT and CE set in the IP header.	0.66
TCPExposure-based MPTCP tests: The client sends a SYN packet containing the *MP_CAPABLE* TCP option to the server; the server inserts the *MP_CAPABLE* option in its SYN/ACK response. This is followed by a TCP data packet from the client containing the *MP_DATA* option, which the server answers using an *MP_ACK* option in its ACK packet.	0.745

Table III: Tests with success ratios (how often packets passed through the network in both directions).

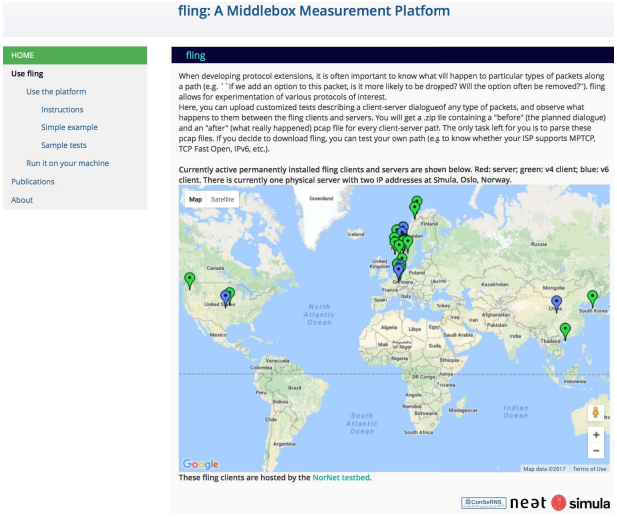

Figure 5: The front page of the *fling* platform

fling, however, is to be able to carry out tests without always having to personally interact with users—we made *fling* simple on purpose to increase the chance of convincing users to install it as a permanent piece of software or deploying it in other measurement platforms.

To this end, we have already begun to develop a *fling* platform that follows the described usage scenario. Figure 5 shows a screenshot of the front page.[4] The world map shows currently active "permanent *fling*" hosts, which are provided to us by the NorNet testbed.[5] NorNet nodes are multi-homed, meaning that a marker on the map translates into 2-3 different IP addresses, connected to different networks (around 4-6 for IPv6-capable hosts, which are shown in blue). These hosts are clients, configured to pull and run the current set of tests every hour. For now, there is only one server, in Oslo, Norway; however, we are planning to use many other nodes as servers as well (as part of the initial handshake with the Fling Control Channel, the main server can inform a "permanent *fling*" client about other servers that it should carry out its tests with).

After registering, users can run their own tests on the *fling* platform. To do this, a user designs a json test description and a pcap file containing the packets used in the dialogue. Instructions are provided on the website; the easiest way to create a test is by downloading and editing one of our (currently 44) examples, which also are the tests that *fling* already runs by default. Next, the user clicks "Use the platform", where (s)he can login using her/his credentials. This leads to a private space where users can upload a test, run it and obtain the result (typically after 1-2 hours). The output is provided as a zip file that contains client- and server-side pcap files for every path that was tested. We are currently developing an auto-generated summary text file to be included in the zip file, containing more information about the test.

[4]This page is available at http://fling-frontend.nntb.no. Here, we will also make the source code of all *fling* components available, and we are planning to share anonymized datasets as well as a number of select permanent measurement results in order to identify long-term trends.

[5]https://www.nntb.no

B. Scaling up

To make our platform grow in scale, we see three major requirements that we need to satisfy:

1. Making it attractive to use: The website must make it clear that *fling* is useful and easy to handle. In addition to offering instructions and example tests, we have therefore created a page ("Simple example" in the menu) that lets users interactively test *fling* on the spot, from the browser. The page shows a diagram with a 3-way TCP handshake followed by a data packet sent over TCP; when a user clicks the IPv4 and TCP headers in the dialogue, the headers are shown with many editable fields. Next to the dialogue, a map shows the locations of the two special clients that we have assigned to participate in this interactive test; one of them is behind a NAT. After changing header fields at will, a user can press "Run the test", wait for a few seconds and obtain output that shows which packets were dropped and/or fields have changed. Our "one-time *fling*" client is also available for download. It is easy to run and produces an immediate output with interesting facts about the user's own Internet connection.

2. Making it attractive to install: The "permanent *fling*" client for end users is currently under development. It will be secure: by definition, *fling* initially only corresponds with our own trusted server, which is the only server that is allowed to redirect a client to other servers (users however need to accept that arbitrarily "strange" packets are transmitted between their client and *fling* servers; we will inform users about this as they download the permanent client). It will be possible to limit the total number packets that the client is willing to send and receive per hour (to limit overall load) and per second (to prevent *fling* tests from interfering with the user's other traffic). We have discussed more security aspects in section III. To make the tool itself appealing, we are considering different options on how the client should present itself (e.g. a user interface that provides information about the current connection's state and informs users about recent "special" tests). Additionally, as the platform grows, we may introduce a credit system like the RIPE Atlas [43] or Seattle [49] to only allow users to upload new tests if they have run the permanent client for some time.

3. Ensuring scalability: Our server in Oslo serves as a trusted entry point to the system; it is also the place where we collect measurement results. This makes our server the most critical element in the infrastructure. *fling* is lightweight; because all tests descriptions are pulled from the server, a simple server update suffices to limit the number of packets or the length of a *fling* measurement (and we can impose such limits on tests that users upload). We will also implement a measure to ensure that the communication with *fling* servers is spread out in time. Instead of requiring servers to remember clients and schedule their access time, we plan to implement a distributed collision avoidance strategy from sensor networks,

where communication also happens at regular intervals but these intervals can differ between clients [15].

fling began with our wish to have more informed discussions in the Internet Engineering Task Force (IETF): we believe that it would be fantastic if theories on what middleboxes would do to certain protocols or header fields could spontaneously and convincingly be tested by uploading a test description to a platform, waiting for a few hours and getting a result. This would require a platform that is as flexible and easy to use as *fling* already is, but larger. Having covered the "flexibility and ease of use" requirement, our immediate next step is to increase the scale of the platform in accordance with the plans that we have laid out. The recent creation of the IRTF Research Group on Measurement and Analysis for Protocols (MAPRG) shows that the IETF is certainly interested in the more informed dialogue about middlebox behavior that we envision.

VI. ACKNOWLEDGEMENTS

This work was partially funded by the European Union's Horizon 2020 Research and Innovation Programme through A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) project under Grant Agreement no. 644334. The views expressed are solely those of the authors.

REFERENCES

[1] "http://openhip.sourceforge.net."
[2] "http://packetlife.net."
[3] "https://redmine.openinfosecfoundation.org."
[4] "https://wiki.wireshark.org."
[5] "https://www.samknows.com."
[6] "http://tcpreplay.appneta.com/."
[7] "http://www.secdev.org/projects/scapy."
[8] "http://www.spiceupyourknowledge.net/2012/11/decrypting-esp-packet-using-wireshark.html."
[9] V. Bajpai and J. Schönwälder, "A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, 2015.
[10] F. Baker, "Requirements for IP Version 4 Routers," RFC 1812 (Proposed Standard), Jun. 1995.
[11] R. Barik, M. Welzl, and A. Elmokashfi, "How to Say That You'Re Special: Can We Use Bits in the IPv4 Header?" in *ANRW '16*, 2016.
[12] S. Bellovin, "The Security Flag in the IPv4 Header," RFC 3514 (Informational), Internet Engineering Task Force, Apr. 2003.
[13] A. Botta and A. Pescap, "Monitoring and measuring wireless network performance in the presence of middleboxes," in *WONS 2011*, Jan 2011, pp. 146–149.
[14] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413 (Experimental), Internet Engineering Task Force, Dec. 2014.
[15] R. L. Cigno, M. Nardelli, and M. Welzl, "SESAM: A semi-synchronous, energy savvy, application-aware MAC," in *WONS 2009*, Feb 2009.
[16] R. Craven, R. Beverly, and M. Allman, "A Middlebox-cooperative TCP for a Non End-to-end Internet," in *SIGCOMM '14*, 2014, pp. 151–162.
[17] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing Middlebox Interference with Tracebox," in *IMC '13*. ACM, 2013.
[18] S. Dhesikan, D. Druta, P. Jones, and C. Jennings, "DSCP Packet Markings for WebRTC QoS," Internet Engineering Task Force, Internet-Draft draft-ietf-tsvwg-rtcweb-qos-18, Aug. 2016, Work in Progress.
[19] K. Edeline, M. Kühlewind, B. Trammell, E. Aben, and B. Donnet, "Using UDP for Internet Transport Evolution," ETH, Tech. Rep. ETH TIK Technical Report 366, Dec. 2016.
[20] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-Start for TCP and IP," RFC 4782 (Experimental), Internet Engineering Task Force, Jan. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4782.txt
[21] R. Fonseca, G. M. Porter, R. H. Katz, S. Shenker, I. Stoica, R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "IP options are not an option," University of California, Berkeley, Tech. Rep. UCB/EECS-2005-24, 2005.
[22] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An Experimental Study of Home Gateway Characteristics," in *IMC '10*, 2010.
[23] R. Hay and W. Turkal, "TCP Option to Denote Packet Mood," RFC 5841 (Informational), Internet Engineering Task Force, Apr. 2010.
[24] M. Honda, "Lessons Learnt from Middlebox Measurement," in *Proceedings of IETF-93, Presentation to HOPS RG*, 2015.
[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *IMC '11*. ACM, 2011.
[26] D. Katz, "IP Router Alert Option," RFC 2113 (Proposed Standard), Internet Engineering Task Force, Feb. 1997.
[27] S. Kent, "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108 (Historic), IETF, Nov. 1991.
[28] ——, "IP Authentication Header," RFC 4302 (Proposed Standard), Internet Engineering Task Force, Dec. 2005.
[29] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the Edge Network," in *IMC '10*, 2010, pp. 246–259.
[30] M. Kühlewind, S. Neuner, and B. Trammell, "On the State of ECN and TCP Options on the Internet," in *Proceedings of PAM'13*, 2013, pp. 135–144.
[31] I. R. Learmonth, B. Trammell, M. Kühlewind, and G. Fairhurst, "PATHspider: A Tool for Active Measurement of Path Transparency," in *ANRW '16*, 2016.
[32] A. M. Mandalari, M. Bagnulo, and A. Lutu, "Informing Protocol Design Through Crowdsourcing: the Case of Pervasive Encryption." ACM SIGCOMM Workshop on Crowdsourcing and crowdsharing of Big (Internet) Data (C2B(I) D), Aug. 2015.
[33] S. McQuistin and C. S. Perkins, "Is Explicit Congestion Notification Usable with UDP?" in *IMC '15*. ACM, 2015, pp. 63–69.
[34] A. Medina, M. Allman, and S. Floyd, "Measuring Interactions Between Transport Protocols and Middleboxes," in *IMC '04*, 2004, pp. 336–341.
[35] ——, "Measuring the Evolution of Transport Protocols in the Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.
[36] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)," RFC 7401 (Proposed Standard), Internet Engineering Task Force, Apr. 2015.
[37] A. Müller, F. Wohlfart, and G. Carle, "Analysis and Topology-based Traversal of Cascaded Large Scale NATs," in *HotMiddlebox '13*, 2013.
[38] J. Padhye and S. Floyd, "Identifying the TCP Behavior of Web Servers," in *In ACM SIGCOMM*, 2000.
[39] J. Pahdye and S. Floyd, "On Inferring TCP Behavior," in *SIGCOMM '01*, 2001.
[40] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *USENIX NSDI'12*, 2012, pp. 29–29.
[41] P. Srisuresh, B. Ford, S. Sivakumar, and S. Guha, "NAT Behavioral Requirements for ICMP," RFC 5508 (Best Current Practice), Internet Engineering Task Force, Apr. 2009.
[42] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations," RFC 2663 (Informational), Internet Engineering Task Force, Aug. 1999.
[43] R. N. Staff, "RIPE Atlas: A Global Internet Measurement Network," *Internet Protocol Journal*, vol. 18, no. 3, Sep. 2015.
[44] S. Sundaresan, S. Burnett, N. Feamster, and W. de Donato, "BISmark: A Testbed for Deploying Measurements and Applications in Broadband Access Networks," in *USENIX ATC 14)*, Jun. 2014, pp. 383–394.
[45] B. Trammell and M. Kühlewind, "Observing Internet Path Transparency to Support Protocol Engineering," in *Proceedings of IRTF/ISOC RAIM Workshop*, Oct 2015.
[46] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An Untold Story of Middleboxes in Cellular Networks," in *SIGCOMM '11*, 2011.
[47] N. Weaver, C. Kreibich, M. Dam, and V. Paxson, "Here Be Web Proxies," in *Proceedings of PAM '14*, 2014.
[48] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan, "Investigating Transparent Web Proxies in Cellular Networks," in *PAM '15*, 2015.
[49] Y. Zhuang, A. Rafetseder, and J. Cappos, "Experience with Seattle: A Community Platform for Research and Education," in *2013 Second GENI Research and Educational Experiment Workshop*, March 2013, pp. 37–44.