Good Cop, Bad Cop: Forcing Middleboxes to Cooperate

Costin Raiciu, Vladimir Olteanu, Radu Stoenescu University Politehnica of Bucharest

1. INTRODUCTION

The original Internet architecture offered a clean contract to endpoints: packets sent will be delivered unmodified [1] or dropped when there is congestion. The proliferation of middleboxes has broken this simple contract to the point where the service the Internet provides to endpoints is entirely unpredictable:

- Reachability depends on fields in the packet header and even the payload, as firewalls strive to contain increasing levels of malicious traffic targeted at vulnerable endpoint software.
- Packets can be modified en-route by boxes that understand the higher level protocol (either TCP or app-level) and optimize it. For instance, NATs support FTP by rewriting the IP address of the sender inside the TCP payload to match the address of the NAT. As the NAT's IP address will likely have a different length, this forces NATs to also modify sequence and acknowledgment numbers. Other performance enhancing middleboxes are discussed in [1].

Firewalls not only drop all unknown protocols or extensions (e.g. SCTP [7], ECN [6]), but they also constrain reachability for traditional protocols: there is no guarantee that UDP or TCP outside ports 80/443 work through many networks including office, cellular or hotspots [3].

This pushes most apps to rely on tunneling to reliably get through networks. HTTP is a favourite amongst mobile apps, and it has even been touted as the new hourglass of the Internet [5]. However, tunneling adds framing overhead and the effect is quite pronounced when the tunneled traffic is UDP-like (e.g. VOIP): in such cases (useless) retransmissions and head-of-line blocking increase jitter and degrade app-performance. A minority of apps use adaptive tunneling to ensure reachability and the smallest possible overhead: for instance, Skype tries UDP, then TCP and finally HTTP or HTTPS. This approach is also suboptimal: certain middleboxes rate limit UDP tunnels to a level where Skype can check reachability but can't make calls [2].

Content-modifying middleboxes are more problematic: they optimize for known apps (e.g. FTP/HTTP) but can break apps that utilize the same port numbers as the known apps. For instance, HTTP parsers can reply with cached contents instead of forwarding the request to the server which can break end-to-end semantics of apps tunneling over HTTP. Because of this, apps are forced to tunnel over HTTPS, thus hiding their traffic from the operator. This outcome is suboptimal for all parties: mobiles spend more energy to encrypt and decrypt traffic (15% in our tests on a Galaxy Nexus) and the operator can't see the traffic anymore and can't protect its customers and network against attacks.

Content modifying middleboxes also increase complexity in new protocols. Multipath TCP [2], a TCP extension that allows using multiple paths in a single TCP connection, includes a *redundant* checksum in the DSS option to ensure it can function correctly with content-changing middleboxes: when a change in payload detected MPTCP either closes the affected subflow or reverts to plain TCP if the affected subflow is the only available one.

In this paper we discuss two fundamentally different directions in which we can improve the status quo. Our first proposal is a "good cop": we present an API endpoints can use to understand the contract offered by the network, and discusses how one might implement this API. We find a few win-win scenarios where both the mobile and the operator gain from collaborating, but it is arguable whether these are enough for this API (or one similar) to be deployed. Our second proposal is a "bad cop" and proposes that all apps use a multitude of tunnels at all times: it treats middleboxes as adversaries and tries to render ineffective any policing they do, while still offering efficient communication. We believe that our good cop-bad cop approach can force network operators to setup their middleboxes such they behave better to the endpoints in the near future.

2. GOOD COP: AN API TO ALLOW COLLABORATION BETWEEN ENDPOINTS AND THE NETWORK

Assume the network operator and the endpoint (apps) want to collaborate. What is an appropriate API that would solve the problems we've outlined so far? The API must offer primitives that allow endpoints query about reachability and packet modifications, while at the same time allowing the operator to maintain its internal network topology and configuration private. To alleviate privacy concerns, our approach aims to allow queries whose answer the clients can find out anyway by using active probing. The usefulness of the API is that it gives definite answers quickly, rather than having to wait for an arbitrarily long probing process to finish. Our API allows endpoints to ask their network operator questions using the following syntax:

```
reach <node> [flow] -> <node> [flow] [const fields]
```

In the syntax above, `node` describes the source or destination of traffic and can be: an IP address or a subnet, the keyword *client* to denotes subnets of the operator's residential clients, or the keyword *internet* to refer arbitrary traffic originating from outside the operator's network.

The `flow` specification uses *tcpdump* format and constrains the flow that departs from the corresponding node. By altering the `flow` definition between the source and destination nodes we can specify how the flow should be changed between those nodes.

Using this syntax clients can express how they would like the network to behave without actually knowing the network topology or the operator's own policy. For instance, the client below expects that Internet UDP traffic can reach its private IP address on port 1500:

```
reach internet udp -> client dst port 1500
```

The `const` construct allows users to specify invariants: packet header fields that remain constant on a hop between

[1]With the exception of the TTL and checksum fields.
[2]Discussion with Romanian cellular operators.

two nodes. For this, the user adds `const` and the header fields, in tcpdump format, that should be invariant. In the example below, the client specifies that the payload should not be modified in the operator's network:

```
reach internet tcp->client src port 80 const payload
```

The operator's reply to these client questions is binary, indicating whether the property holds for the *client's traffic as it passes through the operator's network*. Note that there is no guarantee the traffic will not be modified outside the network operator's domain; however, most transparent middleboxes today are deployed at the "first-hop" operator while the backbone is mostly middlebox free: by asking their first hop operator the clients can be reasonably sure their requirements hold on the whole end-to-end path.

2.1 Implementing the API

We can implement the API by deploying a controller in the operator's network that accepts requests from authenticated clients. The controller knows the topology of the operator, including a snapshot of router forwarding tables and the deployed middleboxes. In our implementation, the middleboxes are implemented as Click modular router [4] configurations. A configuration is a directed graph of Click elements which are processing units performing a simple task such as decreasing TTL or filtering certain packets. We have manually modeled the behaviour of individual Click elements. To answer client requests we use Symnet, a static analysis tool that applies symbolic execution to networks [8].

The controller runs client reachability checks as follows. It first creates a symbolic packet using the initial flow definition or an unconstrained packet, if no definition is given, and injects it at the initial node provided. The controller uses SymNet to track the flow through the network. The output of SymNet is the flow reachable at every node in the network, together with a history of modifications and constraints applied to each packet field. The controller then checks reachability constraints by verifying that the flow spec provided in a given node matches the one resulting from symbolic execution. The requirement is satisfied if there exists at least one flow (symbolic) that conforms to the verified constraints. To check invariants, the controller simply checks whether the field value was not modified on any possible path between the source and the destination nodes.

SymNet helps find a yes/no answer for every client question, but the operator can go further: if the answer is negative, it can reconfigure its middleboxes dynamically to honour the client's request, and then give a positive answer. The API effectively tells the operator *what* the client wants, an information that is not available today.

2.2 Use cases

Client apps can use the API to quickly decide what protocol to use, including port numbers, as well as deciding whether checksumming is needed. Below we discuss two use cases we have implemented that we believe showcase the usefulness of the API.

Protocol Tunneling. Say we wish to run SCTP (or any other new protocol) over the Internet. Deploying it natively is impossible because middleboxes block all traffic that is not TCP or UDP. Thus SCTP must be tunneled, but which tunnel should we use? UDP is the best choice, but it may not work because of firewalls that drop non-DNS UDP packets. In such cases, TCP should be used, but we expect poorer performance because of bad interactions between SCTP's congestion control loop and TCP's.

SCTP has to be adaptive about the tunnel it uses: first try UDP and fall back to TCP if UDP does not work, but to make the decision we need at least one timeout to elapse at the sender—three seconds according to the spec. Instead, the sender can use the API to send a UDP reachability requirement to the operator network. This request takes around 200ms to answer in our implementation, allowing the client to make the optimal tunnel choice much faster.

HTTP vs. HTTPS. Mobile apps often tunnel their data over HTTP to communicate with their servers because it just works, but application optimizers may alter HTTP headers (e.g. accepted-encoding) or the payload itself (compression), breaking the application's own protocol. Should the applications use HTTPS instead to bypass such optimizers? We have measured the energy consumption of a Galaxy Nexus phone while downloading a file over WiFi at 8Mbps. The download times are almost identical, while the energy consumption over HTTP was 570mW and 650mW over HTTPS, 15% higher. The added cost of HTTPS comes from the CPU cycles needed to decrypt the traffic.

Smaller energy consumption is a strong incentive for mobiles to use HTTP, but this may break apps, so we are stuck with the suboptimal solution of using HTTPS. Instead, the client should send an invariant request to the operator asking that its TCP payload not be modified.

3. BAD COP: NINJA TUNNELING

Our second approach is pessimistic, assuming network operators will not cooperate to alleviate the troubles caused by middleboxes to new endpoint apps. In this case, we propose that endpoints should simply force the network to allow reachability and avoid packet changes by using a technique we call *ninja tunneling*. With ninja tunneling, every endpoint always uses a set of tunnels to get through their first hop network operator, such as cellular, DSL or hotspot. The set of tunnels can dynamically be changed in time, and it can include any from the following non-exhaustive list: native IP, UDP, TCP, HTTP, HTTPS, DNS, covert channels, etc. The set of tunnels is forever changing, perhaps in response to network behaviour.

The key to ensure *ninja tunneling*'s success is that tunnels are invisible to applications: unmodified apps should benefit from it. The ninja tunnels will be deployed as software in the (mobile) operating system and at content/cloud providers that terminate the tunnels initiated by the clients. Should all tunnels originating from a mobile user be terminated by the same cloud machine, or can we use tunnels terminated by different machines spread through the Internet? The latter is more attractive because it would make detection and filtering of ninja tunnels much more difficult.

The answer to the question above is closely tied with another issue: how should application traffic be spread over this dynamic collection of tunnels? In this position paper, we only consider the case when both the endpoint and its remote endpoint (e.g. the server it is receiving a service from) have upgraded to Multipath TCP. Spreading traffic over multiple tunnels is trivial with MPTCP: each tunnel will be treated as a different path and MPTCP will create

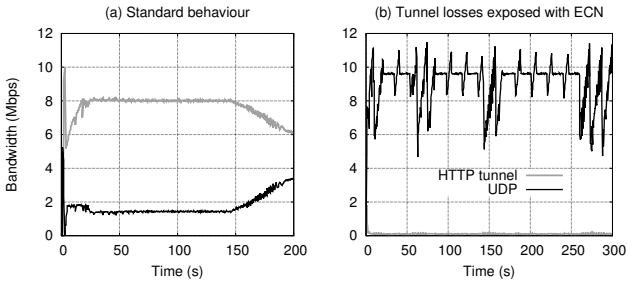

Figure 1: Runnning MPTCP over an UDP and HTTP tunnel sharing a 10Mbps link

a corresponding subflow. If certain tunnels do not work, MPTCP will simply move traffic onto tunnels that do work.

It is plain to see that, as long as there is some sort of connectivity allowed, ninja tunnels will provide reachability to any application. As the set of tunnels can be constantly evolved and personalized per endpoint, network operators will have a difficult time identifying groups of tunnels that are related and an even harder time deciding which ones to drop. We do not claim it is impossible for operators to block such tunnels, it will just be much more expensive for them to do so. Rate-limiting certain tunnels, as done with Skype today, will also not work, as MPTCP will push more traffic through the other tunnels. Finally, the Multipath TCP checksum will also detect payload changes on the different subflows, and MPTCP will terminate the corresponding subflows when changes are detected , therefore ensuring the integrity of the payload.

Efficiency. A natural concern regarding ninja tunneling is efficiency: many of these tunnels (e.g. covert channels) have a high framing overhead and have a poor ratio of useful traffic to total traffic. As the bottleneck in most networks is the user's access link (e.g. cellular connection or DSL line), using a mix of inefficient tunnels will simply decrease the total goodput of the user. Ideally, we would like the endpoint to utilize the most efficient tunnel at all times. Determining this in advance is not possible because network operators could throttle certain traffic after a period of time (e.g. after DPI makes a decision to classify the traffic).

The Multipath TCP congestion controller actively moves traffic away from congested paths onto uncongested paths, as determined by the congestion window of each subflow [9]. Is this mechanism enough to push traffic through the most efficient tunnels? We ran experiments downloading a large file (with wget over MPTCP) over a mixture of the following tunnels, all sharing the same 10Mbps access link connecting our client to the server: UDP, TCP, HTTP, DNS. We found that MPTCP behaved correctly when UDP and DNS tunnels were used, pushing most of the traffic over the more efficient UDP tunnel and achieving a throughput of around 9.5Mbps; the DNS tunnel only achieves 7Mbps.

However, when using TCP-based tunnels (e.g. TCP or HTTP) in conjunction with DNS, MPTCP pushed a lot of traffic over the less efficient TCP tunnel, as shown in the figure 1.a. The problem is that the TCP tunnel hides the losses to the MPTCP congestion controller and increases delay in the process. The effect is that the MPTCP congestion controller does not move traffic away from the TCP tunnel. To avoid this problem, we implemented a technique that sets an ECN mark in the MPTCP subflow whenever the tunnel experiences a loss. This technique achieves its purpose, allowing MPTCP to balance traffic to the most efficient tunnel, as shown in Fig. 1.b.

Next steps. Many details need to be resolved before ninja tunnels are practical. For instance, what if the remote server does not speak MPTCP? What if the traffic is UDP?

4. CONCLUSIONS

We have outlined two diverging proposals that aim to solve the same problem: the service offered by the network has become unpredictable, stifling innovation.

Our more pragmatic solution picks a contract that seems reasonable for endpoints, *ubiquitous reachability and immutable payload*, and enforces it by using MPTCP to spread the data over a mix of different tunnels. The only thing an operator can now do is make it cheaper or more expensive to achieve this contract: the app does not have to worry about the details, as ninja tunneling will dynamically find the most efficient tunnel and send most data through it. Eventually, operators that force endpoints to use inefficient tunnels will lose their customers, so this may well act as a driving force towards removing unwanted middleboxes.

We have also outlined a constructive approach where the endpoints can use an API to query the network about the service it offers. This offers more information to both operators and endpoints, and in the long run we believe this approach is preferable for operators: they still get to run their app optimizers for traffic that allows it explicitly, but they will have to honour the requests of users asking middleboxes to not mess with their traffic.

The API and ninja tunneling complement each other: ninja tunneling is a short term fix and a stick to beat operators into being nice. The API is the proper way of implementing cooperation, but it is doomed without something like ninja tunnels that forces operators to adopt it.

Acknowledgements

This work was partly funded by Trilogy 2, a research project funded by the European Commission (FP7 317756).

5. REFERENCES

[1] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. RFC 3135: Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations, June 2001.

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. RFC 6824: TCP Extensions for Multipath Operation with Multiple Addresses, January 2013.

[3] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. ACM IMC*, 2011.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems, August 2000*, 2000.

[5] L. Popa, A. Ghodsi, and I. Stoica. Http as the narrow waist of the future internet. In *Hotnets*, 2010.

[6] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP , September 2001.

[7] R. Stewart. RFC 4960: Stream Control Transmission Protocol, September 2007.

[8] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Static checking for stateful networks. In *HotMiddlebox*, 2013.

[9] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, 2011.