In-Net : In-Network Processing for the Masses

Radu Stoenescu[‡] Vladimir Olteanu[‡] Matei Popovici[‡] Mohamed Ahmed[*]
Joao Martins[*] Roberto Bifulco[*] Filipe Manco[*] Felipe Huici[*] Georgios Smaragdakis[†]
Mark Handley[•] Costin Raiciu[‡]

[‡] University Politehnica of Bucharest [*] NEC Europe [†] MIT CSAIL [•] University College London
firstname.lastname@cs.pub.ro firstanme.lastname@neclab.eu gsmaragd@csail.mit.edu
m.handley@cs.ucl.ac.uk

Abstract

Network Function Virtualization is pushing network operators to deploy commodity hardware that will be used to run middlebox functionality and processing on behalf of third parties: in effect, network operators are slowly but surely becoming in-network cloud providers. The market for in-network clouds is large, ranging from content providers, mobile applications and even end-users.

We show in this paper that blindly adopting cloud technologies in the context of in-network clouds is not feasible from both the security and scalability points of view. Instead we propose IN-NET, an architecture that allows untrusted endpoints as well as content-providers to deploy custom in-network processing to be run on platforms owned by network operators. IN-NET relies on static analysis to allow platforms to check whether the requested processing is safe, and whether it contradicts the operator's policies.

We have implemented IN-NET and tested it in the wide-area, supporting a range of use-cases that are difficult to deploy today. Our experience shows that IN-NET is secure, scales to many users (thousands of clients on a single inexpensive server), allows for a wide-range of functionality, and offers benefits to end-users, network operators and content providers alike.

1. Introduction

Middleboxes have been deployed in most networks to increase security and application performance, to the point where they are visible on a large fraction of end-to-end paths [19] and are as numerous as routers and switches in enterprise networks [33]. Until recently, middleboxes were rather expensive hardware appliances that were difficult to upgrade and scale and locked in operators to hardware vendors such as Cisco or Juniper.

Network function virtualization (NFV) aims to run the functionality provided by middleboxes as software on commodity hardware, and is being embraced by network providers because it promises to reduce costs for purchasing, upgrading and scaling middleboxes. The transition to NFV has gained a lot of traction in standards groups: there is a NFV group in ETSI [37] and Service Function Chaining has been chartered as a new working group at the IETF to allow sending traffic via multiple network functions. We are also starting to see initial deployments: Deutsche Telekom is deploying racks of x86 servers at its regional POPs [3], and other operators are following suit.

By deploying x86 infrastructure, network operators aim to go further than running cheaper middleboxes: they also want to offer processing on demand to other parties, effectively becoming miniature cloud providers specialized for in-network processing. Such a shift would re-invigorate the business models of operators, faced with dwindling profits from their main business of selling bandwidth.

Content providers are very keen to perform processing in the network, close to the users, to reduce latencies and gain performance: in the past ten years they have done so by relying on CDNs for both static and dynamic content (e.g., Akamai [24, 29]) or by deploying hardware in access networks (e.g., Google Global Cache [1, 14] or Netflix Open Connect [2]). Even end-user applications deploy middleboxes to bypass NATs, as is the case with Skype supernodes or STUN servers. More and more mobile applications require in-network support to achieve inbound connectivity behind provider NATs and personalized traffic optimization to reduce energy consumption (e.g. by batching) and bandwidth usage of cellular links.

There is thus a potentially large market for in-network processing, and network operators are shaping up as the cloud operators for such processing. Today's clouds are based in few locations and host millions of servers. In con-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/10.1145/2741948.2741961

trast, in-network clouds will be highly distributed, with a few racks in each mini-cloud (PoP) and tens to hundreds of mini-clouds per network operator.

To enable in-network clouds, the current belief is that one can just adapt the technologies used in public clouds (e.g. Amazon's EC2) in the context of in-network processing. We observe, however, that this approach is not feasible: current approaches do not scale to a large number of tenants per machine, and operator security or policy requirements may be violated by unconstrained tenant processing. There is a fundamental tension between the programming model offered to tenants (x86 VM in public clouds) and the ability to understand whether the tenant processing is safe to run.

Our contribution is an architecture to enable in-network processing called IN-NET. IN-NET restricts the programming model offered to tenants to Click (as in [27]) and addresses the challenges above by:

- Defining a set of security rules that ensure in-network processing is safe to operators and the Internet.
- Defining an API that allows both operators and users to express their policy.
- Using static analysis tools (symbolic execution) to automatically check whether client processing satisfies provider policy and security requirements, and whether the client's own requirements are met.
- Optimizing ClickOS, a guest operating system that can run Click configurations in Xen virtual machines, to support 1,000 tenants on a single physical machine.

We have built a prototype of IN-NET that encapsulates all these contributions. We ran tests in lab environments to understand its scaling limits (Section 6), and implemented a range of middleboxes to see how accurate static checking is in practice. In addition, we have deployed IN-NET on a wide-area testbed, and implemented a number of different use cases that showcase the architecture's potential (Section 8). Our experiments show that IN-NET scales to large numbers of users and brings performance benefits to end users, operators and content providers alike.

2. Requirements and Approach

Enabling in-network cloud processing raises a number of issues compared to traditional clouds: scale (many more users), security and policy compliance.

Scale. In-network clouds will cater for both businesses and (mobile) end-users, thus the number of tenants of in-network computing is expected to be quite large. It is possible that many of these tenants will process little traffic. As users are mobile and processing needs to follow the user for best performance, we also expect the churn to be much higher: in-network clouds need to promptly handle many more requests to start or terminate processing.

What technology shall we use to enable **in-network processing platforms**? Providing x86 VMs, as used in clouds today, will not scale: Xen can only run tens of VMs with commodity OSes on a single machine, hitting multiple scaling barriers. One limitation is memory: a stripped down Linux VM has a 500MB memory footprint, so supporting processing for one thousand users with one VM per user will need in excess of 500GB, which is wasteful considering the processing done on behalf of each user is typically very simple, and most users are only active for short periods of time.

In-network processing is specialized to packet and flow-processing, and we can use restricted programming models to support it much more efficiently. In previous work it has been shown that it is possible to build a wide range of in-network processing using the Click [23] modular router. The same authors propose ClickOS, a guest operating system optimized to run Click that can reach packet speeds nearing 10Gbps line-rate (14Mpps) while running tens to a hundred of VMs on the same machine [27].

We use ClickOS as the basis of our solution to enable in-network processing for the masses. We enhance ClickOS to scale beyond 100 users by using two novel techniques (§5): *safely* consolidating multiple users on the same VM and instantiating virtual machines on the fly, as packets arrive. To highlight the fact that user processing is not necessarily instantiated as a separate VM, in this paper we term a unit of processing on behalf of a user a **processing module**.

2.1 Security Requirements

Basic security issues inside a processing platform are similar to those in public clouds: virtualization offers both performance and security isolation, and accountability ensures that users are charged for the resources they use, discouraging resource exhaustion attacks against platforms.

Today's clouds rely on firewall appliances to police inbound traffic to their tenants' VMs and typically allow all reasonable outbound traffic (e.g. TCP, UDP). Inbound cloud traffic (mostly HTTP/HTTPS request traffic) is much smaller in size than outbound traffic, making it economical to police. Not policing outbound traffic does have its downsides: there are reports that "trial" accounts are used in public clouds to launch DDoS attacks against other Internet clients. If we applied the same security policy for in-network clouds, the scale of the possible DDoS attacks would be much larger and the threat posed by in-network processing for network operators' standard customers (e.g. home or business Internet users) and the Internet at large would be unacceptable.

In-network clouds, by their distributed nature, must further restrict what their tenants can do, or risk becoming the ultimate network attack tool. We have devised a set of security rules that must be obeyed by in-network tenants, ensuring that such attacks are not possible.

A first requirement is that tenants can only process traffic destined to them, and cannot spy on or illegally interact with other tenants' or operator traffic. To achieve this

requirement, we simply assign unique IP addresses to each tenants' processing module, and ensure that only traffic that is destined to the processing module actually reaches it.

Should all tenants be allowed to generate outbound traffic to any destination? The answer is obviously *no*—otherwise it would be easy to launch DDoS attacks using IN-NET. We cannot easily change the Internet to be default-off [17] and spoofing free [1], but we can make sure that traffic generated by IN-NET platforms obeys these principles. To implement default-off, IN-NET requires that any third-party generated traffic must be authorized by its destination; authorization can be explicit or implicit.

- **Explicit authorization** occurs when either a) the destination has requested the instantiation of the processing or b) the destination is a processing module belonging to the same user. In the first case, the user will keep its network operator updated with a number of addresses that he owns and uses. In the second case, the provider needs to implement a way to disseminate the IP addresses of a certain user to all interested platforms.
- **Implicit authorization** occurs when a host A sends traffic to a processing module B. If B is the destination, then it is implicitly authorized to use A as the destination address of the response traffic; this rule is similar in spirit to existing firewall behavior that allows incoming traffic corresponding to established outgoing connections. This rule allows IN-NET platforms to reply to traffic coming from regular end hosts (e.g., to implement a web server).

To prevent spoofing, IN-NET platforms ensure that traffic leaving a platform has the platform's IP as a source address or the same address as when it entered the platform.

The set of rules above apply when the tenant is an untrusted customer of the in-network cloud service. The network operator's residential or mobile customers are still subject to the anti-spoofing rules, but are also allowed to send traffic to any destination, without the destination explicitly agreeing. This extension is natural: it allows client in-network processing to use the same service that the client itself receive from the operator (Internet connectivity). This implies that such customers can also deploy explicit proxies.

Finally, the operator's own processing modules are allowed to generate traffic as they wish, reflecting the trust the operator places in them. Static analysis only helps the operator decide whether the box is achieving its intended purpose (i.e., correctness, as opposed to security).

2.2 Policy Requirements

Network operators apply local policy to customer traffic: they prioritize certain traffic classes, apply application specific optimizations (e.g. HTTP middleboxes) and check for security threats (Intrusion Detection Systems/firewalls). When in-network clouds take off, traffic originated by the tenants must obey the same policy as all other traffic, but applying policy is not as simple anymore, as the operator would need to know beforehand what type of traffic the tenant will generate before choosing a location to run it.

Consider the in-network cloud provider shown in Figure 3, whose policy dictates that all HTTP traffic follow the bottom path and be inspected by the HTTP middlebox. If a client's VM talks HTTP, it should be installed on Platform 2 so that that the traffic can be verified by the middlebox. Installing the client's VM on Platform 1 would disobey the operator's policy.

[1] All attempts to date have failed to gain any traction.

3. Static Analysis as Basis for In-network Processing

Before in-network processing becomes a reality, there are also some practical hurdles: (i) how can operators mitigate the risks of running third-party services in their core networks? and (ii) how can potential customers be sure operators handle their traffic and processing appropriately?

IN-NET overcomes this hurdle by enabling operators and endpoints **to reason about the operation of middleboxes on their traffic, before instantiating the traffic or processing**. To do so, IN-NET treats the network (the endpoint, middleboxes and paths between them) as distributed program, and the packets it carries as the variables of this program. In this model, tracing the path a given packet takes in the network is akin to modeling the control flow graph of a program: IN-NET applies static analysis techniques from compilers (in particular symbolic execution) to analyze the operations of the network with respect to the traffic it carries.

To understand how this works, consider the example in Figure 1. The network operator runs a stateful firewall that only allows outgoing UDP traffic, and related inbound traffic. A mobile customer wants to tunnel its application data over UDP to a server in the Internet, and needs the payload to travel unchanged all the way to the server. Assuming the client knows the pseudocode of the firewall (given in Figure 2.a), it could create a program snippet to model the network configuration, and check whether its payload is unchanged:
`firewall_out(client)`

To apply symbolic execution, a symbolic packet is created in the client code. A symbolic packet represents a set of packets with certain common constraints (e.g., all packets sourced by one host), and consists of a set of variables that model header fields and the payload. Each variable can be free or bound to a symbolic expression (another variable, set of variables, or set of possible values). Free variables can take any value, while bound variables express restrictions on the values of the corresponding field.

In the example above, the packet fields (IP source, destination, protocol and data) are set to the symbolic variables `CLI`, `SRV`, `PROTO` and `DATA`, all of which are free, capturing traffic from all clients to any Internet destination. The symbolic packet leaves the client and is fed as input to

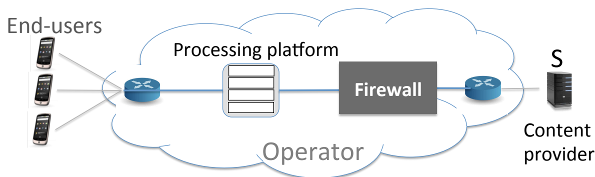

Figure 1. In-network processing example topology

the `firewall_out` routine, which leaves existing packet fields unchanged, but restricts the value of `p[proto]` to UDP. By comparing the value of `p[data]` after the firewall to the value before it, the client notices they point to the same symbolic variable D. Thus, the data will not change en-route to the server, and it will arrive as long as it is sent over UDP.

Checking Operator Policy Compliance. Now consider a more complex scenario, also in Figure 1. Assume the server S responds to customers with the same packet, by flipping the source and destination addresses, as shown in the pseudocode given in Figure 2.a. S wants to run its processing in the operator's platform to get lower latency. Is there a risk that the provider's clients will be attacked by S's in-network processing code?

One way to reassure the operator is to sandbox the processing at runtime. However, sandboxing is not only expensive (33%-70% drop in throughput in our evaluation), but also imprecise. A better way is to use static analysis. To understand whether its policy is obeyed, the provider can apply symbolic execution in two configurations: the original setup where the server runs somewhere in the Internet, and **platform**, where the server runs in the operator's network.

The symbolic execution trace for the first case is shown in Figure 2.b, where the symbolic packet headers are shown at the top of the first table, and their bindings are shown below these as we run reachability on the code shown in Figure 2.a. On every line, shaded cells denote changes in value. For instance, when the packet exits the client, its source address is set to unbound variable `CLI` and the destination to `SRV`. In the firewall, `p[proto]` is constrained to have value "UDP". The symbolic packet reaching the client is the last line of the trace. Running symbolic execution on the **platform** setup yields exactly the same symbolic packet, implying the two configurations are equivalent. Hence, it is safe for the operator to run the content-provider's server inside its own network, without sandboxing.

4. The In-Net Architecture

Our discussion so far has highlighted that static analysis can help to overcome the issues raised by in-network processing, namely fear of running un-trusted code and understanding interactions between customer traffic and operator processing. However, we have made some major assumptions:

```
client():
 p = create_packet();
 p[ip_src] = CLI;
 p[ip_dst] = SRV;
 p[proto] = PROTO;
 p[data] = DATA;
 return p;
```

IP SRC	IP DST	PROT	DATA	FW TAG
?	?	?	?	
CLI	?	?	?	
CLI	SRV	?	?	
CLI	SRV	P	?	
CLI	SRV	P	D	

```
firewall_out(Packet p):
 if (p[proto]==udp)
   p[firewall_tag]=true;
   return p;
 else return NULL;
 //packet dropped
```

IP SRC	IP DST	PROT	DATA	FW TAG
CLI	SRV	udp	D	
CLI	SRV	udp	D	true

```
server(Packet p):
 if (p[proto] == "UDP")
   old_dst = p[ip_dst];
   p[ip_dst] = p[ip_src];
   p[ip_src] = old_dst;
   return p;
 else return NULL;
 //packet dropped
```

IP SRC	IP DST	PROT	DATA	FW TAG
CLI	SRV	udp	D	
CLI	SRV	udp	D	true
CLI	CLI	udp	D	true
SRV	CLI	udp	D	true

```
firewall_in(Packet p):
 if (p[firewall_tag])
   return p;
 else return NULL;
//packet dropped
```

IP SRC	IP DST	PROT	DATA	FW TAG
SRV	CLI	udp	D	true

(a) Network model (b) Symbolic execution

Figure 2. Static checking of the configuration in Figure 1. The code is shown on the left-hand side, and the corresponding symbolic execution trace is shown on the right.

1. It is possible to model middleboxes in a way suitable for static checking. This is not a given, since middleboxes are arbitrary code and inferring the needed model could prove intractable.
2. The clients know what operator-deployed middleboxes do. This clearly violates the operator's need to keep its network policy private, and would become a major stumbling block to adoption.

Our goal is to embed static analysis as the basis of a network architecture that is deployable today, and that offers benefits to network operators, content providers and customers alike. The crux of our proposal is that access network operators become miniature cloud providers, with a focus on running in-network functionality for themselves, their clients, or paying third parties.

To implement our architecture, operators will deploy a controller that receives requests from clients and instantiates processing on one of the **processing platforms** installed in their network (see Figure 3). The controller knows the operator's topology and policy and uses a symbolic execution tool called SYMNET [35] to statically check client requests.

To address assumptions 1 and 2, we built an API for clients able to express the processing desired in a way that is a) amenable to automatic static checking, b) flexible enough to capture a multitude of in-network processing requests and c) allows clients to check interactions with operator policies

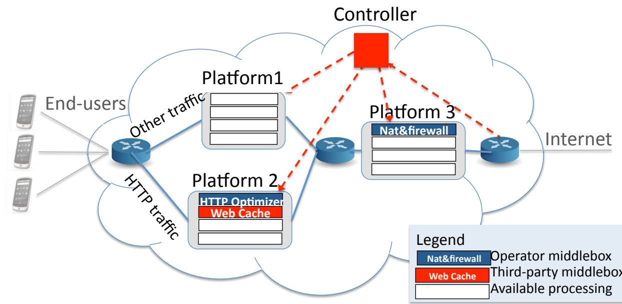

Figure 3. In-Net architecture: access operators deploy processing platforms where they run their own middleboxes and processing for third parties. A controller installs client processing after checking it for safety.

without knowing those policies. We next discuss each of these items in detail, and wrap up with an example that shows how the components work together.

4.1 The API

In-network processing is rather domain-specific - both input and output are network traffic. Most operations applied to packets are simple and include NATs, filters, tunneling, proxies, shaping, forwarding and so on, and arbitrarily complex functionality can be built by combining these basic building blocks and by creating new ones.

As a result, IN-NET clients express processing requests by using an extended version of the configuration language used by the Click modular router software [23]. The Click language has the notion of *elements*, small units of packet processing such as `DecIPTTL`, `Classifier`, `IPFilter` (and hundreds of others) which users interconnect into graphs called *configurations*.

A key advantage of using modular programming is that we can automatically analyze the client's processing as long as it relies only on known elements.

Client Requests contain two parts: (1) the *configuration* to be instantiated and (2) the *requirements* to be satisfied (Figure 4). In an In-Net platform, a *processing module* is the unit of processing, and it is the result of the instantiation of a client-provided configuration. The configuration can essentially take two forms: either a Click configuration using well-known Click elements (please refer to the Click modular router information for more details about Click elements and their syntax) or a pre-defined "stock" processing modules offered by the platform.

Stock modules can range from Click configurations to software running on commodity OSes. Our prototype controller, for instance, offers a reverse-HTTP proxy appliance, an explicit proxy (both based on squid), a DNS server that uses geolocation to resolve customer queries to nearby replicas, and an arbitrary x86 VM where customers can run any processing. The latter offers flexibility at the cost of security and understanding of the processing: such VMs are sandboxed at runtime and are more expensive to run (please refer to Section 7.2 for more details).

```
Batcher module:
  FromNetfront() ->
  IPFilter(allow udp port 1500) ->
  IPRewriter(pattern - - 172.16.15.133 - 0 0)
  -> TimedUnqueue(120,100)
  -> dst::ToNetfront()

reach from internet udp
-> Batcher:dst:0 dst 172.16.15.133
-> client dst port 1500
   const proto && dst port && payload
```

Figure 4. Client request with a single processing module for simple UDP port forwarding.

Requirements allow both operators and clients to express their policy, and we discuss them in detail next.

4.2 Requirements

Our policy language must be able to specify traffic reachability for certain traffic classes between different parts of the network, including routing via way-points. Additionally, as packets are modified by middleboxes, we need a way to specify what modifications are allowed.

Expressing network policy has traditionally been done using VLANs to provide isolation, ACLs for access control, and policy routing to specify routing not based on the destination address. This results in specifications that are difficult to compose and verify, so we cannot use them.

Novel policy languages have been proposed in the context of OpenFlow to describe operator policy. FML is a declarative language that allows operators to use a simple syntax to express predicates about groups of users, and specify how these groups can communicate [18]. FlowExp is a lower level policy language proposed in [22] where routes can be specified exactly or with wildcards. FlowExp also allows specifying constraints (exact values or wildcards) for packet headers. FlowExp comes fairly close to our needs, however it cannot capture the transformations made by middleboxes to packet headers: it is not possible to specify, for instance, that a certain header field should not be changed.

Our policy language is inspired by these two works. The API supports reachability checks that have the form:

```
reach from <node> [flow]
{-> <node>[flow][const fields]}+
```

In the reachability check, `node` describes a vertex in the network graph and can be:

- An IP address or a subnet.
- The keyword *client* that denotes subnets of the operator's residential clients.
- The keyword *internet* refers to arbitrary traffic originating from outside the operator's network.

- A port of a Click element in a processing module (or port 0, if none is provided).

The `->` denotes traffic that will flow from source to destination. The `flow` specification uses *tcpdump* format and constrains the flow that departs from the corresponding node. By altering the `flow` definition between two nodes we can specify how the flow should be changed between those nodes, thus capturing the "allowed" middlebox processing on that segment.

Clients use requirements to 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 in Figure 4 expects that Internet UDP traffic can reach its private IP address on port 1500. This is stated by:

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

The statement above allows the operator to do any processing as long as *some* UDP packets reach port 1500 of the client. To specify that the Internet traffic is also destined to port 1500, the client needs to add `dst port 1500` to the first line. The client could also request that Internet packets are destined to port 2250; with such a configuration, Internet packets will only arrive at the client if the operator changes the destination port using some processing in its network.

The client can also specify that the UDP traffic goes through Click element dst of processing module $batcher$, on port 0, and, at that point, the IP destination should have been changed to the client's private IP:

```
reach from internet udp
-> batcher:dst:0 dst 172.16.15.133
-> client dst port 1500
```

To specify that a header field remains constant, the client can specify values for it at different hops, as discussed above; however, in certain cases the client does not know apriori the value of the specific header (i.e. source port). Our last construct allows users to specify **invariants**: packet header fields that remain constant on a hop between two nodes. For this, the user adds `const` to the header fields, in tcpdump format, that should be invariant. In the example in Figure 4, the client specifies that the destination port, protocol number and payload must not be modified on the hop from the $batcher$ processing module to the client.

Operators use the same API to describe their policy. Take for instance the operator in Figure 3. It can express that all HTTP traffic reaching clients must go through the HTTP Optimizer as follows:

```
reach from internet tcp src port 80
-> HTTPOptimizer
-> client
```

4.3 The In-Net Controller

The job of the controller is to take client requests and statically verify them on a snapshot of the network; this snapshot includes routing and switch tables, middlebox configurations, tunnels, etc. Our controller relies on SYMNET, a symbolic execution tool tailored for networks [35].

It is well known that symbolic execution does not scale well when run on complex programs (e.g. involving loops, dynamic memory allocation), despite the fact that existing symbolic execution tools for C programs have made strides into what is possible to symbolically execute [9]. Middleboxes are complex code, and the fact they keep per-flow state makes the verification task even harder.

To avoid these issues, IN-NET does not check arbitrary code: it only deals with Click configurations composed of known elements, and it checks *abstract models of the middleboxes* rather than the middlebox code. SYMNET runs symbolic execution over Haskell descriptions of the network elements, and it scales well because our abstract models are written to enable fast verification [2]:

- They contain no loops.
- There is no dynamic memory allocation
- Middlebox flow state is modelled by pushing the state into the flow itself, which allows SYMNET to be oblivious to flow arrival order.

Our controller is implemented in Scala. It parses Click configurations and outputs Haskell code that SYMNET can verify. We have manually modeled all the stock Click elements. At startup, the controller is provided with the operator's network configuration including available platforms and their network addresses, routers and snapshots of their routing tables. Next, the controller is provided with a set of rules (using the API above) that must always hold. The policy is enforced by static verification performed by the controller at each modification of the state of the network (i.e., Click module deployment), as described next.

The controller runs a SYMNET reachability check for each requirement given. 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. SYMNET then tracks the flow through the network, splitting it whenever subflows can be routed via different paths, and checking all flows over all possible paths. For each header in the symbolic packet, SYMNET tracks restrictions on its current value, and also remembers the last definition of this header field (when a node last modified it). The output of SYMNET is the flow reachable at every node in the network, together with a history of modifications and constraints applied at each hop.

[2] More information about the abstract models used by SYMNET can be found in [35].

Using this information, the IN-NET controller checks reachability constraints by verifying that the flow specification 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 header field was not defined on the last hop. For instance, in our example in Figure 4, the invariants for fields proto, destination port and payload hold because the corresponding header fields in the symbolic packet are not overwritten on the hop between $batcher$ and the client.

Scaling the controller. In our prototype, the IN-NET controller runs on a single machine and our evaluation in section 6.1 shows it can answer queries in under a second for fairly large networks; such delays are acceptable because they only occur when users processing modules to be installed, and not during packet processing. However, if the load on the controller increases or the network is larger, the controller may become a bottleneck. Fortunately, most of the decisions the controller makes are about correctness and installing individual processing elements only affects that user's traffic; we conjecture it is fairly easy to parallelize the controller by simply having multiple machines answer the queries. Care must be taken, however, to ensure requests of the same user reach the same controller (to ensure ordering of operations), or to deal with problems that may arise when different controllers simultaneously decide to take conflicting actions: e.g. install new processing modules onto the same platform that does not have enough capacity.

Client configuration. Before sending processing module deployment requests to the IN-NET controller, the client installs IN-NET software locally and configures it with the client's credentials, and with the address of the controller. We assume this address is obtained by out-of-band methods (e.g., register with the network operator, as in clouds today).

The client can then submit its processing requests to the controller. The controller statically verifies if there exists a platform where the client's processing module can run and adhere to both operator and client policies. To do so, it iterates through all its available platforms, pretends it has instantiated the client processing, checking all operator and client requirements. If no platform is found, deployment is not safe and the client is notified accordingly.

If a matching platform is found, the Click module is deployed on the chosen platform and is assigned a client-unique identifier. The client is also given an IP address, protocol and port combination that can be used to reach that module. The controller then alters the operator's routing configuration: at the very least, it will install forwarding rules on the target platform to ensure that the processing module receives traffic destined for the IP address/protocol/port combination. In our implementation, we use Openflow rules to configure Openvswitch running on each platform for this purpose. Finally, clients can stop processing modules by issuing a kill command with the proper identifier.

4.4 Checking Security Rules

Enforcing security policies relies on a mix of static analysis and, if necessary, sandboxing.

For all clients, the controller verifies the client's policy, but also ensures that the client does not spoof traffic. To check against spoofing, the controller injects an unconstrained symbolic packet (i.e. any possible traffic input) into the processing module and observes the output. The source address of traffic leaving the platform must either be the address assigned to the client by the controller, or the source address header field should be invariant along any path from the entry to the exit of the module.

For untrusted third-parties, the controller must further ensure the destination agrees with receiving that traffic. In this case, the controller checks the destination address of packets originating from the processing module. Conforming packets will obey one of the following rules:

- The destination address is in a white-list that is maintained per-client; or
- The destination address is equal to the source address of the incoming traffic.

Checking the first rule is trivial. Checking the second rule is possible because of the power of symbolic execution: SYMNET will process the definitions of the destination field such as $IP_{dst} = IP_{src}$ by marking that IP_{dst} is bound to the same unknown variable IP_{src} was bound to when it entered the platform, and hence the invariant holds.

If the rules are satisfied, the module is guaranteed to be safe, granted individual Click elements are bug-free. If the rules are not satisfied, there are two possible outcomes: (i) either all the traffic is not conforming, and thus the processing module should not be run, or (ii) the module can generate both allowed and disallowed traffic, and compliance cannot be checked at install time.

In the second case, the faulty behavior may not triggered in real life, thus leading to false positive alerts: the malicious flows exposed by symbolic execution may not occur during the functioning of the module (e.g., a tunnel is one such case). In this case, the operator will run the processing module in a sandbox.

Sandboxing. Arrakis [31] or IX [8] propose to use hardware virtualization features in modern NICs that include rate-limiting and filtering/white-listing to enable safe I/O pass-through to virtual machines. Reusing the same mechanisms to sandbox IN-NET clients is not possible: hardware support is not available for stateful rules such as implicit authorization in IN-NET. Hardware support can be used for our static security rules, however existing NICs only support tens to a few hundred rules, limiting the number of users supported on a single box.

We rely on software sandboxing instead. We have implemented a new Click element called `ChangeEnforcer` that sandboxes one processing module. Its implementation resembles that of a stateful firewall: it allows traffic from the outside to the processing module, and only related response traffic from the processing module to the outside world. The sandboxing element is additionally configured with a whitelist of addresses the processing module is allowed to access.

When a processing module requires sandboxing, the controller transparently instantiates an instance of the enforcer for each interface of the processing module. This instance is added to the client configuration when a Click configuration is being run; this has the benefit of billing the user for the sandboxing. The element will be injected on all paths from the `FromNetfront` element to elements in the processing module, and on all paths from the processing module to the `ToNetFront` element.

When the processing module is arbitrary code (x86 VM), the sandboxing runs in a separate VM on the same machine, and traffic is properly directed to the sandbox using Openflow rules. As expected, our evaluation shows this option is more expensive from a processing point of view.

4.5 A Unifying Example

We will use a concrete example—push notifications for mobiles— to illustrate how to use the IN-NET architecture. Such notifications are useful for many apps including email, social media and instant messaging clients, and today require the mobile device to maintain a long-running TCP connection to a cloud server with keep-alives (roughly every minute) to ensure that NAT state does not expire. Apps wanting to notify the mobile can do so via the cloud server. Unfortunately, this mechanism allows applications to send as many messages as they wish, keeping the device's cellular connection awake and draining its battery much quicker [6].

We can solve this problem with IN-NET. Assume the mobile customer wants to batch incoming UDP notifications on port 1500 for UDP traffic. The client request is shown in Figure 4: traffic received on port 1500 by the processing module is forwarded to the client's IP address. Additionally, the module restricts the IPs that can send notifications, and batches traffic to save energy. Upon receiving this request from a customer, the controller: (1) Finds a suitable platform to instantiate the requested processing. At every potential platform it uses SYMNET to see if both the provider's and the customer's requirements can be met. In the example in Figure 3, only Platform 3 applies, since Platforms 1 and 2 are not reachable from the outside. (2) If there are suitable platforms, it instantiates the processing; otherwise, it does nothing. In the example given the processing module is started on Platform 3. (3) Finally, it informs the client of the outcome (the external IP address of the processing module in this case).

5. The In-Net Platform

IN-NET platforms are based on Xen and so inherit the isolation, security and performance properties afforded by paravirtualization. As such, the platforms can run vanilla x86 VMs, though this is not our main target because it severely limits the scalability of the system. For the platform to be viable, it has to be able to support a potentially large number of concurrent clients while ensuring isolation between them.

Consequently, the IN-NET platforms rely on ClickOS [26], a guest operating system optimized to run Click configurations. ClickOS consists of a tiny Xen virtual machine built from combining MiniOS (a minimalistic OS available in the Xen sources) and the Click modular router software [23]. ClickOS supports a large range of middleboxes such as firewalls, NATs, load balancers, and DNS proxies, with many more possible thanks to the over 200 Click elements built into the ClickOS VM image; for more on ClickOS please see [26]. Here, we focus on modifications and experimentation we carried out to build a viable IN-NET platform around ClickOS.

Scalability via on-the-fly middleboxes. ClickOS virtual machines are tiny (5MB when running), allowing us to run up to 100 of them on inexpensive commodity hardware as explained below. Even so, an In-Net platform is likely to have to service many more clients. One key observation is that since ClickOS VMs can boot rather quickly (in about 30 milliseconds), we only have to ensure that the platform copes with the maximum number of *concurrent* clients at any given instant. Thus, we transparently instantiate ClickOS VMs *on-the-fly*, when we see packets destined to a client configuration that is not already running.

To achieve this, we modify ClickOS' back-end software switch to include a switch controller connected to one of its ports. The controller monitors incoming traffic and identifies new flows, where a new flow consists of a TCP SYN or UDP packet going to an In-Net client. When one such flow is detected, a new VM is instantiated for it, and, once ready, the flow's traffic is re-routed through it.

Suspend and resume. Creating VMs on the fly works great as long as the client's processing is stateless or only relevant to the single flow the VM was created to handle. For stateful handling, and to be able to still scale to large numbers of concurrent clients, we add support to MiniOS to allow us to suspend and resume ClickOS VMs; we present evaluation results for this in Section 6.

Scalability via static checking. The one-client-per-VM model is fundamentally limited by the maximum number of VMs a single box can run, and on-the-fly instantiation mitigates the problem but is no panacea. We could of course further increase capacity with servers with large numbers of CPU cores, or use additional servers, but this would just be throwing money at the problem.

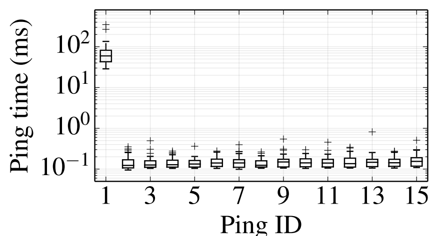

Figure 5. ClickOS reaction time for the first 15 packets of 100 concurrent flows.

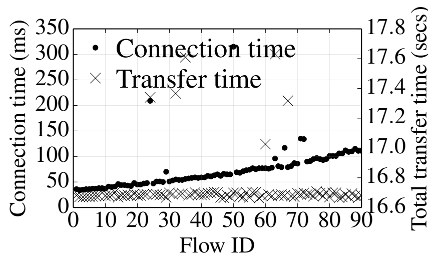

Figure 6. 100 concurrent HTTP clients retrieving a 50MB file through a IN-NET platform at 25Mb/s each.

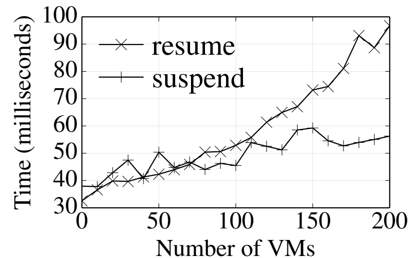

Figure 7. The x-axis shows the number of existing VMs when one more VM is suspended or resumed.

It is better to run multiple users' configurations in the same virtual machine, as long as we can guarantee isolation. To consolidate multiple users onto a VM we create a Click configuration that contains all user configurations preceded by a traffic demultiplexer; no links are added between different users' configurations, and no elements instances are shared. Explicit addressing ensures that a client's module will only see traffic destined to it, and our security rules ensure that processing modules cannot spoof IP addresses. Standard Click elements do not share memory, and they only communicate via packets. This implies that running static analysis with SYMNET on individual configurations is enough to decide whether it is safe to merge them.

For Click elements that keep per-flow state, ensuring isolation is trickier: one user could force its configuration to use lots of memory, DoS-ing the other users. To avoid such situations we would need to limit the amount of memory each configuration uses. Our prototype takes the simpler option of not consolidating clients running stateful processing.

6. Evaluation

IN-NET has two major components: its scalable platform software, and the controller. In this section we evaluate the scalability of these two components.

Platform Scalability. To estimate the number of clients a powerful platform might support, we first ran basic experiments on one of our more expensive servers, a machine with 4 AMD Opteron 6376@2.3GHz processors (64 cores in total) and 128GB of RAM. In these experiments, we simply booted up as many virtual machines as we could, and we ran both Linux and ClickOS VMs. We were able to run upto 200 stripped down Linux VMs, each with a 512MB memory footprint. In comparison, the memory footprint of a ClickOS VM is almost two orders of magnitude smaller (around 8MB), and we were able run as many as 10000 instances of ClickOS on the same hardware.

These numbers are upper bounds, but we would like to know how many concurrent users can share an In-Net platform while carrying out actual middlebox processing on modest hardware, which we believe is more representative of practical deployments. First we test ClickOS's ability to quickly react to incoming traffic by instantiating on-the-fly virtualized middleboxes. We connect three x86 servers in a row: the first initiates ICMP ping requests and also acts as an HTTP client (`curl`), the middle one acts as the IN-NET platform and the final one as a ping responder and HTTP server (`nginx`). For the actual processing we install a stateless firewall in each ClickOS VM. All measurements were conducted on a much cheaper (about $1,000), single-socket Intel Xeon E3-1220 system (4 cores at 3.1 GHz) with 16 GB of DDR3-ECC RAM running Xen 4.2.0.

In our first experiment we start 100 pings in parallel, with each ping sending 15 probes. Each ping is treated by the platform as a separate flow, and a new VM is started when the first packet is seen. Figure 5 shows the results: the first packet in a flow experiences higher delays because of the overhead of VM creation, but its round-trip time is still only 50 milliseconds on average. For subsequent packets the ClickOS VM is already running and so the RTT drops significantly. The RTT increases as more ClickOS VMs are running, but even with 100 VMs the ping time for the first packet of the 100th flow is 100 ms. When running the same experiment with stripped-down Linux VMs, the average round-trip time of the first packet is around 700ms which is an order of magnitude higher than that of ClickOS and unnaceptable for interactive traffic, e.g. web browsing.

Next, we conducted a more realistic experiment where each client makes an HTTP request capped at 25Mb/s. The requests go through an In-Net platform in order to reach an HTTP server hosting a 50MB file, and the client processing (plain forwarding) is booted when the SYN arrives. We measured the time it takes for the connection to be set up (i.e., including VM creation) as well as the total time to transfer the file, and plot the results in Figure 6 (the connection times are longer than the transfer ones because the files are relatively small).

Suspend and resume. Starting and terminating ClickOS VMs is ideally suited for stateless network processing, such as a plain firewall. When VMs hold per-flow state, however, terminating a VM would effectively terminate the end-to-end traffic, which is unacceptable. The solution in this case is to use suspend/resume instead of terminate/boot. To this

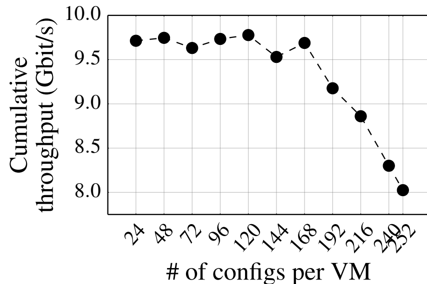

Figure 8. Cumulative throughput when a single ClickOS VM handles configurations for multiple clients.

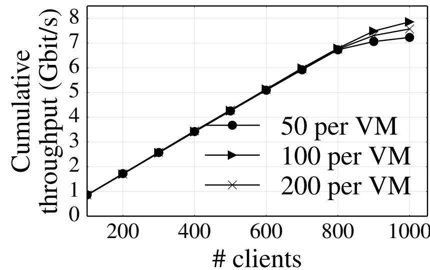

Figure 9. Throughput when a box has up to 1,000 clients with different number of VMs and clients per VM.

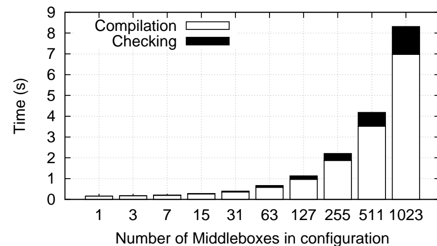

Figure 10. Static analysis checking scales linearly with the size of the operator's network.

end, we have implemented suspend and resume support in MiniOS and performed experiments varying the number of existing VMs when we suspend and resume a single VM. The results are shown in Figure 7 highlighting that it is possible to suspend and resume in 100ms in total.

Aggregating multiple users onto a single virtual machine. Another way to scale the number of clients served by a IN-NET box is by *consolidating* the processing modules of different clients onto a single ClickOS virtual machine, with proper demultiplexing and multiplexing of traffic (the consolidation is safe as long as the processing is stateless). To understand the scalability of this optimization, we create a single ClickOS VM and install a Click configuration containing an `IPClassifier` element to act as a destination IP demuxer. For each client we run individual firewall elements, and then traffic is again multiplexed onto the outgoing interface of the VM.

We start up an increasing number of HTTP flows, one per client, and measure the cumulative throughput through our platform (Figure 8). As shown, we can sustain essentially 10Gb/s line rate for up to over 150 clients or so, after which the single CPU core handling the processing becomes overloaded and the rate begins to drop. While the exact point of inflexion will depend on the type of processing and the CPU frequency, these results show that aggregation is an easy way to increase the number of clients supported by a platform.

Finally, we gradually increase the number of clients up to 1,000 by running multiple VMs and different numbers of clients per VM (n=50, 100 or 200). Each client is downloading a web-file at a speed of 8Mbps and the n'th client triggers the creation of a new VM. We measure the cumulative throughput at the IN-NET platform and plot the results in Figure 9. Even with all VMs pinned to a single core, the platform yields 10Gbps throughput in this experiment, highlighting its scalability to many clients.

MAWI traces. Is 1,000 clients a realistic target? To understand whether IN-NET can scale to real-world workloads, we downloaded and processed MAWI traces [4] (the WIDE backbone in Japan) taken between the 13th and 17th of January 2014. Each trace covers 15 minutes of traffic, and we eliminate all connections for which we do not see the setup and teardown messages. Most of the traffic we ignore is background radiation (instances of port scanning), but some of it is due to longer connections intersecting the 15-minute trace period. The results show that, at any moment, there are at most 1,600 to 4,000 active TCP connections, and between 400 to 840 active TCP clients (i.e., active openers). The exact thresholds varies with the day of the week, but the main take-away is that a single IN-NET platform running on commodity hardware could run personalized firewalls for all active sources on the MAWI backbone.

6.1 Controller Scalability

We now turn to the controller: how long does it take to respond to a user's processing request? We ran an experiment where the client issued the request shown in figure 4. For the provider topology in figure 3, the server needs 101ms to compile the Haskell rules that our front-end generates, and just 5ms to run the analysis itself; this implies that static checking can be used for interactive traffic.

To understand how this number scales to a more realistic operator topology, we randomly add more routers and platforms to the topology shown in figure 3 and measure the request processing time. The results in Figure 10 show that static processing scales linearly with the size of the network. The biggest contributor to the runtime is the time needed to compile the Haskell configuration. In practice, the operator will have a compiled version of its configuration, and we only need to compile the client's configuration and load it dynamically. SYMNET scales well: checking reachability on a network with 1,000 boxes takes 1.3 seconds.

Modeling stateful middleboxes—such as a stateful firewall—has similar performance. This is because SYMNET and verification and memory costs grow linearly with the number of stateful boxes encountered. The abstract models of Click elements that we use for verification avoid state explosion, unlike model-checking based techniques such as AntEater [25] or when symbolic execution is run on actual code [12].

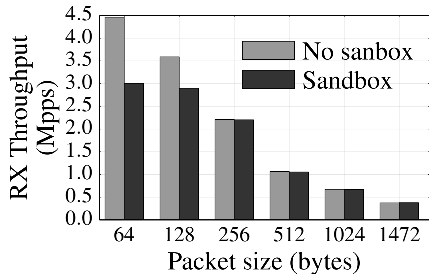

Figure 11. Cost of sandboxing in a IN-NET platform.

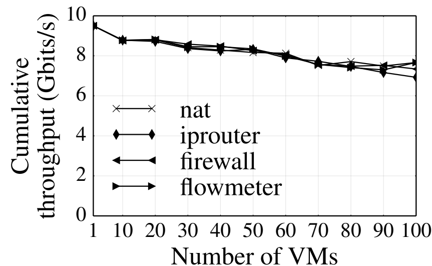

Figure 12. IN-NET platforms run many middleboxes on a single core with high aggregate throughput.

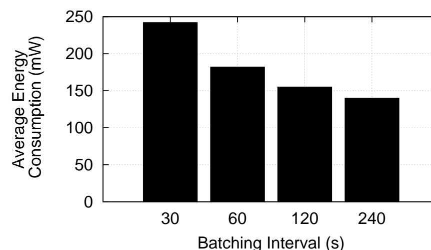

Figure 13. Mobiles save energy when an In-Net platform batches push traffic into larger intervals.

7. Security

The design of the IN-NET architecture aims to strike a balance between flexibility and security. Adding flexibility to the ossified Internet is paramount, and we achieve this by allowing untrusted third-parties to request and instantiate in-network processing. Avoiding abuse is done by ensuring processing modules do not spoof traffic, and adhere to the default-off principle [17]: processing platforms can only send traffic to destinations that want to receive that traffic.

To implement default-off, we need to ensure that destinations explicitly agree to receive traffic and that this agreement cannot be forged by malicious third parties. If these conditions hold, we are guaranteed that processing platforms cannot themselves be used to launch Denial of Service Attacks against endpoints [17]. Additionally, IN-NET processing can also be used to defend against traffic originating from existing hosts or misbehaving platforms: destinations can instantiate filtering code on remote platforms, and attract traffic to those platforms by updating DNS entries.

There are however two possible caveats with the DDoS protection offered by IN-NET, and these allow amplification attacks and time-based attacks, which we discuss next.

The first caveat regards forging authorization to send traffic by exploiting the *implicit authorization* rule: an attacker can send packets to a processing module using packets with spoofed source addresses. This implicitly (and fakely) authorizes the processing module to communicate with the traffic source. This attack can be used to launch traffic amplification attacks prevalent today with DNS: the attacker sends a low rate of DNS requests with the spoofed source address of the victim, and the DNS servers amplify the attack by sending large DNS response packets [5].

The operators can mitigate such attacks by applying ingress filtering on both the Internet links and its client links. This will ensure that operator's clients can only attack other clients, and that users in the Internet can only attack other users in the Internet, and not the operator's clients. To completely eradicate amplification attacks the operators can ban connectionless traffic (e.g. UDP): amplification attacks are not possible with TCP because the attacker cannot complete the three-way handshake. In fact, operators must choose between flexibility of client processing and security.

The second caveat is that SYMNET does not model time: implicit authorization, once given, allows the processing module to send traffic until an explicit connection teardown is observed (for TCP) or forever (UDP). In practice, stateful firewalls authorize response traffic only for a certain period of time; when the connection is idle longer than a timeout value, authorization is revoked (this is how the `ChangeEnforcer` element works). We believe this attack is tolerable in practice.

Routing Policy Enforcement. With IN-NET, destinations have ultimate control over the path taken by incoming packets, allowing them to implement routing policy as needed using a mix of traffic attraction functions such as tunnels, and NATs. It has recently been shown that a single tunnel is enough to offer increased reliability and protection against DDos attacks [30].

7.1 Coverage of Static Checking

Symbolic execution can give false positive answers, implying that a configuration breaks the security rules when in fact it does not. To understand how accurate SYMNET checking is, we implemented a range of different middleboxes using existing Click elements or by deploying IN-NET stock processing modules. We then used the controller to check these for safety. In Table 1 we show the middleboxes we have implemented, and the assessment provided by SYMNET on their safety. The answers depend on who is requesting the processing, as the security constraints are different for third-party providers, clients or the operator itself. Inaccurate answers are marked (s) and require sandboxing.

SYMNET gives accurate answers for most of these middleboxes, with two exceptions. The need to check x86 VMs is obvious, however it is interesting to discuss the tunnel scenario. Here, the security rules only allow the third-party customer to send traffic to a predefined list of destinations, but the actual destination address of packets is provided at runtime, when packets are decapsulated at the processing module. SYMNET finds that the processing module might send

Functionality	Third-party	Client	Operator
IP Router	✗	✗	✓
DPI	✗	✗	✓
NAT	✗	✗	✓
Transparent Proxy	✗	✗	✓
Flow meter	✓	✓	✓
Rate limiter	✓	✓	✓
Firewall	✓	✓	✓
Tunnel	✓(s)	✓	✓
Multicast	✓	✓	✓
DNS Server (stock)	✓	✓	✓
Reverse proxy (stock)	✓	✓	✓
x86 VM	✓(s)	✓(s)	✓

Table 1. Running SYMNET to check middlebox safety gives accurate results.

traffic to legitimate addresses, thus it cannot just deny the client's request. On the other hand, the client could reach destinations it should not, hence the need for sandboxing.

7.2 Sandboxing

How costly is sandboxing? We use a single ClickOS VM to receive traffic directly via a 10 Gbps NIC or through our `ChangeEnforcer` sandboxing element (recall Section 2.1) inserted in the Click configuration.

Figure 11 shows that the throughput drops by a third for 64B packets, by a fifth for 128B packets, and does not experience any measurable drop for other packet sizes. Sandboxing with the enforcer running in a separate VM, in comparison, is much more expensive. Throughput for 64B packets drops to 1.5Mpps because of repeated context switching between the processing module VM and the sandboxing VM.

Today's status quo is to run x86 VMs in sandboxes. Our evaluation shows the total throughput of the system drops by 70% because of sandboxing. Luckily, sandboxing is not needed in the first place since we can statically check whether the processing is safe for most client configurations.

8. Use Cases

We now discuss a variety of use-cases showcasing the benefits that our architecture provides to the main actors of the Internet ecosystem: operators, end-users and content-providers. We do not claim these use-cases are novel; all we want to show is that they can now be accessible not only to big content and CDN providers, but to a much wider range of organisations and end-users. The selection of use-cases is also geared towards highlighting that IN-NET is flexible despite its stronger security guarantees.

Software Middleboxes have become prominent with the advent of NFV, and IN-NET offers a safe and scalable way to run many of them on low-end commodity hardware. We deploy a number of different middleboxes on an In-Net platform and measure the aggregate throughput of the box. Traffic is generated by a client running `curl` connected via a 10 Gbps link to the platform, itself connected via a similar link to a server running `nginx` and serving content from a ramdisk. We vary the number of middleboxes on the same core, split the client traffic evenly, and plot the aggregate throughput. Figure 12 shows that the IN-NET platform manages to sustain high throughput, regardless of the number and type of middleboxes.

Push Notifications. We run the configuration presented in Figure 4 using as client a Samsung Galaxy Nexus mobile phone connected via 3G to the Internet. It takes around 3s for the whole request to be executed by the IN-NET controller, which finds the proper placement for the processing module, and checks its security, the clients' requirements and the operator's (this time is dominated by the time needed to wake up the 3G interface). The reply specifies the IP address of the newly allocated processing module. From one of our servers, we send one UDP message with 1KB payload every 30s to the given address and port; the platform batches the messages in the processing module and delivers them at different intervals.

We measured the device's energy consumption with the Monsoon power monitor (Figure 13). Batching has a massive effect on average energy consumption, reducing it from 240mW to 140mW. In this use-case, IN-NET benefits both the client and the cellular operator: the client can trade increased delay in receiving notifications for lower energy consumption, while the operator gets the opportunity to monitor and perhaps filter malicious messages.

Protocol Tunneling. Consider the task of running 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.

To understand the effect of such interactions, we use `iperf` to measure bandwidth between two servers connected via an emulated wide-area link with capacity 100Mbps and a 20ms RTT. We also introduce random losses to understand how the protocol fares under congestion scenarios. The results in Figure 14 show how SCTP over TCP encapsulation dramatically reduces the throughput achieved: when loss rate varies from 1% to 5%, running SCTP over a TCP tunnel gives two to fives times less throughput compared to running SCTP over a UDP tunnel.

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 could use the IN-NET API to send a UDP reachability requirement to the network. This request takes around 200ms, after which the client can make the optimal tunnel choice much faster.

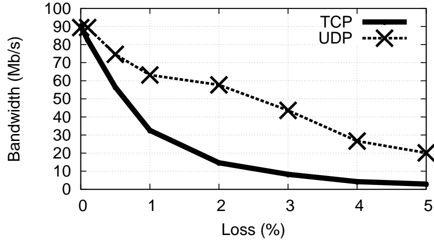

Figure 14. SCTP performance when tunneling over TCP and UDP

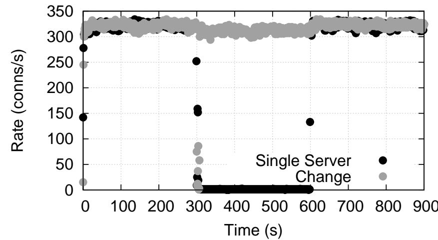

Figure 15. Defending against a Slowloris attack with IN-NET

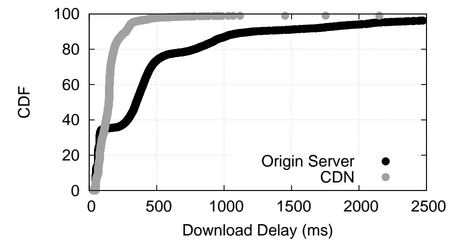

Figure 16. Clients downloading a 1KB file from the origin or our CDN.

HTTP vs. HTTPS. Mobile apps heavily rely on HTTP to communicate to their servers because it just works, and in many cases they are even tunneling other traffic over HTTP. Application optimizers deployed by the network may alter HTTP headers (e.g. accepted-encoding), breaking the application's own protocol. Should the applications use HTTPS instead to bypass such optimizers? We have measured the energy consumption of our Galaxy Nexus 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.

DoS Protection. We use the reverse-proxy stock processing module to defend against an HTTP attack tool called Slowloris that attempts to starve all valid clients of a server by maintaining as many open connections to that server as possible. Slowloris trickles the HTTP request bytes at a very slow rate, which prevents the server from timing out the connection. The best known defense is to just ramp up the number of servers, and this is what we do with IN-NET. When under attack, the web server starts a number of processing modules at remote operators, and redirects new connections via geolocation to those servers. In Figure 15 we plot the number of valid requests serviced per second before, during, and after the attack. We can see that IN-NET is able to quickly instantiate processing and divert traffic, thus reducing the load on the origin server.

Content Distribution Network. IN-NET can safely run legacy code in sandboxed processing modules. As our final use case we run a small-scale content-distribution network as follows. The origin server is located in Italy, and there are three content caches (located in Romania, Germany and Italy) instantiated on IN-NET platforms. Each content cache is an x86 virtual machine running Linux and Squid, the latter configured as a reverse-proxy. The IN-NET controller instantiates sandboxing for such machines using the mechanism described in Section 2.1.

We use 75 PlanetLab nodes as clients scattered around Europe, and we use geolocation to spread them to the caches approximately evenly. We run repeated downloads of 1KB files from all these clients, and we report the results in Figure 16. We see that the CDN is behaving as expected: the median download time is halved, and the 90% percentile is four times lower.

9. Related Work

The idea of in-network processing is not new. Our work builds upon a rich literature on programmable networks, including active networks [10, 36, 39], network operating systems [11, 15], and network virtualization [7, 34].

In active networks packets carry code that routers execute, and it has been shown how such flexibility can be used to support multicast and mobility in IP. While sharing the goal of adding flexibility to the network, IN-NET aims to allow flexibility only at a few points in the Internet, rather than at all routers as in active networks—this choice, together with ongoing deployments of commodity hardware by operators, ensure IN-NET is deployable. Also, active networks focus on individual packets as the processing unit, and thus cannot support middleboxes that maintain flow state. IN-NET focuses on flow processing instead.

More recently, Sherry et al. [33] proposed to migrate middleboxes from enterprises to public clouds to reduce the cost of deployment and maintenance, addressing a number of problems such as secure communication and attraction of traffic to middleboxes in the cloud. While the goals of IN-NET are similar to these works, our paper differs in two key aspects: (1) we believe middleboxes equate to innovation, and want access operators to offer them as a service to third parties, and (2) we show how to use static analysis as a basis for an in-network processing architecture.

More recently, researchers have proposed incorporating middleboxes into the Internet architecture in the Delegation Oriented Architecture (DOA) [38] and NUTSS[16]. The key findings are twofold: first, explicit connection signaling is needed to negotiate connectivity through firewalls, which ensures policy compliance and network evolvability as it decouples the control traffic needed to setup the connection

from the data traffic (which could use a different protocol). Second, middleboxes should only process traffic explicitly addressed to them. We adopt both principles in IN-NET.

Building scalable software middleboxes has been the focus of much research recently. Routebricks [13] shows how to run a scalable router on a cluster of commodity servers. CoMb [32] is an architecture for middlebox deployments that systematically applies the design principle of consolidation, both at the level of building individual middleboxes and managing a network of middleboxes. Finally, IN-NET platforms extend ClickOS [26] by using static analysis to enable it to scale to thousands of client configurations on a single machine.

Statically checking network routing is a well-established topic, with reachability analysis as well as loop detection the strong candidates for verification [40]. More recently, Header Space Analysis (HSA) [20, 21] proposed a more general version of network static analysis that can also model arbitrary middleboxes as transformations of packet headers from input to output. HSA could be used to check IN-NET configurations too, but it has two important shortcomings. First, HSA does not model middlebox state and therefore cannot capture the behavior of common middleboxes such as stateful firewalls or NATs. Secondly, HSA cannot capture invariants. Recently, Dobrescu et al. [12] used dynamic symbolic execution to build a tool that checks middleboxes for bugs, and applied it to binary Click elements; this work is complementary to ours.

Proof Carrying Code [28] has been proposed to allow safe execution of active networking code. PCC takes low-level assembly code and generates automatic proofs that certain properties hold for the code in question; these proofs can be easily verified by third parties. Unfortunately, PCC has only been shown to work with fairly simple programs (e.g. no loops) and applying it to arbitrary middlebox code is not feasible, because automatically building proofs for arbitrary programs is a difficult open problem. Our approach of using static analysis is similar in spirit to PCC; we rely on abstract models of middlebox code (Click elements) and symbolic execution to make the verification task tractable.

10. Conclusion

Network Function Virtualization has pushed network operators to deploy commodity hardware in their networks. These will be used to run middlebox functionality and processing on behalf of third parties: network operators are slowly but surely becoming in-network cloud providers.

In this paper, we have shown that directly adopting cloud technologies in the context of in-network processing is not possible. Instead, we propose to restrict the programming model offered to tenants and use static checking to ensure security and policy requirements are met. We have designed a novel API that allows clients to express their requirements, and implemented it using symbolic execution.

Our solution cheaply ensures security and compliance for arbitrary Click configurations provided by customers, and supports a wide range of use cases relevant to mobile applications, small content providers and even end-users. We have implemented a processing platform for in-network processing that can safely support to 1,000 users on the same hardware. Our experiments show that this platform should be sufficient to implement personalized firewalls for every client of the MAWI backbone link.

By relying on static analysis, IN-NET makes it possible for clients and network operators to reason about network processing without resorting to active probing as is common today [19]. This could provide a basis to elegantly solve the ongoing tussle between network operators and endpoints that, so far, has lead to an ongoing war of encapsulation and Deep Packet Inspection.

Acknowledgements

This work was partly funded by Trilogy 2, a research project funded by the European Commission in its Seventh Framework program (FP7 317756). We also thank the anonymous reviewers and our shepherd Derek Mc Auley for their insightful feedback.

References

[1] Google Global Cache. `http://ggcadmin.google.com/ggc`.

[2] Netflix Open Connect. `https://signup.netflix.com/openconnect`.

[3] A10 Networks. Deutsche Telekom TeraStream: A Network Functions Virtualization (NFV) Using OpenStack Case Study. `http://www.a10networks.com/resources/files/A10-CS-80103-EN.pdf#search=%22management%22`, 2013.

[4] MAWI Working Group Traffic Archive. `http://mawi.wide.ad.jp/mawi/`.

[5] US Cert DNS Traffic Amplification Attacks. `https://www.us-cert.gov/ncas/alerts/TA13-088A`.

[6] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, D. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying Online While Mobile: The Hidden Costs. In *CoNEXT*, 2013.

[7] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM*, 2006.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *OSDI*, OSDI'14, 2014.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. In *IEEE Com. Magz.*, 1998.

[11] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[12] M. Dobrescu and K. Argyraki. Software Dataplane Verification. In *NSDI*, 2014.

[13] M. Dobrescu, N. Egi, K. Argyraki, B. G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *SOSP*, 2009.

[14] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[15] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *Computer Communication Review*, 2008.

[16] S. Guha and P. Francis. An End-Middle-End Approach to Connection Establishment. In *SIGCOMM*, 2007.

[17] H. Ballani and Y. Chawathe and S. Ratnasamy and T. Roscoe and S. Shenker. Off by Default! In *HotNets*, 2005.

[18] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *WREN*, WREN '09, 2009.

[19] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *IMC*, 2011.

[20] P. Kazemian, M. Chang, H. Zeng G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

[21] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[22] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, nsdi'13, 2013.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F.Kaashoek. The Click modular router. *ACM Trans. Computer Systems*, 18(1), 2000.

[24] T. Leighton. Improving Performance on the Internet. *CACM*, 2009.

[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[26] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[27] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr 2014. USENIX Association.

[28] George C. Necula. Proof-carrying code. In *POPL*, POPL '97, 1997.

[29] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS'10*, 2010.

[30] Simon Peter, Umar Javed, Qiao Zhang, Doug Woos, Thomas Anderson, and Arvind Krishnamurthy. One tunnel is (often) enough. In *SIGCOMM*, SIGCOMM '14, 2014.

[31] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, OSDI'14, 2014.

[32] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. The Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[33] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[34] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[35] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Static checking for stateful networks. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, pages 31–36, New York, NY, USA, 2013. ACM.

[36] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *CCR*, 26(2), 1996.

[37] ETSI Network Functions Virtualisation. `http://www.etsi.org/technologies-clusters/technologies/nfv`.

[38] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, 2004.

[39] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *OpenArch'98*, 1998.

[40] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *INFOCOM*, 2005.