Automated Failure Identification under IPv6 Transition Mechanisms

Vaibhav Bajpai, Nikolay Melnikov, and Jürgen Schönwälder
Computer Science
Jacobs University Bremen
{v.bajpai, n.melnikov, j.schoenwaelder}@jacobs-university.de

***Abstract*---The allocation of the last block of IPv4 addresses, an explosive growth of smart phone market throughout past decade, and the on-going process towards the Internet of Things act as some of the main factors for an increasing need for the IPv6. While IPv6 can readily expand the extent of the Internet, deploying it alone is clearly not a solution, due to lack of IPv6 support both by network service providers and current applications running over IPv4. It is anticipated that there will be a long period of co-existence of IPv4 and IPv6 technologies, with a continuum of solutions that will help the Internet to grow. There is a multitude of solutions that promote IPv4 and IPv6 co-existence, but they need to be thoroughly tested and on a large scale, before they can actually be considered as viable solutions. In the following study we present a real-life evaluation of the two popular transitioning technologies: NAT64 and Dual-stack lite. The goal is two-fold: first - identify how well do current applications, protocols and on-line services inter-operate with the aforementioned technologies and detect the potential failures; second - use a stream-oriented flow query language F in order to define a query that could find those failures by scanning through traces of NetFlow formatted data. We established that there are several applications that fail under NAT64, and we show they way we detect these failures in an automated fashion.**

Keywords: NAT64, Dual-stack lite, automated failure detection in IPv4-to-IPv6 transition mechanisms

I. Motivation

The IETF began work on IPv6 in the mid 1990s realizing that it wouldn't take long for the IPv4 address space to run out, given the pace of Internet adoption and growth. By the mid 1990s IPv6 was completely defined, but it did little to displace IPv4 as the de facto standard. This has been due to the lack of any economic advantage for the network providers to deploy IPv6 and a lack of IPv6-only killer applications that can aggressively push the need for end users to look for any service benefit to make the switch. Studies show that IPv6 today accounts for less than 1% of global Internet traffic and only 0.15% of the top 1 million websites are reachable over IPv6 [1].

Today, the issue of depleting IPv4 addresses has become more imminent and undeniable. APNIC recently announced that it had reached the final stage of IPv4 exhaustion in the Asia Pacific region to mark the release of all IPv4 addresses in its available pool [2]. This occurred only few months after IANA allocated the last blocks of IPv4 addresses to the Internet Registries exhausting its pool of unallocated IPv4 addresses [3]. Despite the foresight on the rapid decline, IPv6 is still in the very early stages of deployment. Very few network operators, even those with aggressive deployment plans, have completed IPv6 roll outs. Most network operators have not even begun IPv6 deployment. Usually two factors are attributed to such reluctance: the amount of currently available IPv4 content and potentially large number of applications, both at home and at enterprise networks, that do not work over IPv6, which will make the transition from IPv4 to IPv6 to take multiple years. Therefore, the service providers and the enterprises will either need to better leverage the current pool of addresses by sharing the IPv4 addresses among many customers using wider/layered NAT deployments or employ technologies and mechanisms that allow to reach IPv4 content from IPv6-only hosts during the extended migration phase.

The findings of this study demonstrate that in reality many of the existing applications and protocols inter-operate well with the existing transition mechanisms. These findings correspond to some of the observations made in other independent studies [4], [5]. There are of course, as we found out, several applications that do not work, and these are the "border-line cases" that many operators and service providers are afraid of. These failures can significantly decrease users' overall experiences or satisfaction with the provided services, and would definitely hurt the reputation of the network operator or the service provider. Therefore, the main goal of our study was to identify those border-line cases, detect which applications do not inter-operate with the two popular transition technologies. Once the failing applications were detected, we tried to investigate the trace that these failures leave. In our case - a trace is a sequence of NetFlow-formatted flow records that represent a sequence of network events that occurred in the process of certain application's failure. We term such sequence of flow-records as a *failure signature*. A signature that is unique (or almost unique) for each of the failing applications, since each of them performs a different sequence of network-related operations in the background (at the start up or during the operation). For instance, one of the failing applications that we will investigate in more detail in Section VI, tries to initially find other Skype clients on a local host, after which it proceeds to contact the Skype Login Sever in a very specific manner. These sequence of events is reflected in flow records. Once we understand and define what a failure signature of a certain application is, we can use that signature to automatically scan through massive amounts of flow records

in order to automatically detect the most frequent failures. We see a great benefit in such approach, since it allows application developers, service providers and network operators to detect application failures associated with deployment of different transition technologies very early in the deployment process. It will allow service providers and network operators to take a pro-active rather than reactive approach to problems that users may face after deployment of these transition mechanisms, and increases users' overall satisfaction with the provided service. Besides, an automated approach to *failure signature* identification can provide a broader perspective on the reasons for certain failures, detection of the most common problems and shorten the overall transition mechanism deployment verification cycle.

However, definition of such involved signatures is a rather complex task, which requires non-standard NetFlow processing tools. Since most of the existing flow-processing tools are oriented on absolute filtering of flow records (where a flow record is checked for presence of a single or a collection of certain field values), we turned to use a stream-oriented query processing language F [6], [7], which was developed by our group and had a number of real-life applications to date. The main reason for creation of this query language was to provide users with the possibility of defining complex queries that would reflect a relative and parallel (on the flow record level) nature of many present-day application signatures.

The structure of the paper is as follows. In the next section we discuss the operation of Dual-stack lite (DS-lite) and NAT64. Section III presents the NetFlow data format and overviews the stream-oriented flow query language F. The inter-operability of different applications with DS-lite and NAT64 mechanisms are presented in sections IV and V respectively. In section VI we analyse the reasons for failures of different applications and present the way to define these *failure signatures* in F. Other transition technologies that can potentially be investigated for their inter-operability with different applications are presented in section VII. The paper is concluded in section VIII.

II. Investigated IPv4-to-IPv6 Transition Technologies

In order to perform a more exhaustive evaluation of the IPv4-to-IPv6 transition technologies we set the goal of concluding an in-depth rather than a broad investigation. The goal was to increase the number of the variety of tested applications at the same time limiting the number of transition technologies to just two. The selection has fallen on two technologies that have a wider resonance in the scientific community. As such we selected to investigate: Dual-stack lite and NAT64.

A. Dual-stack lite

Dual-stack lite [8] is an approach that uses IPv6-only links between the provider and the customer. When a device in the customer network sends an IPv4 packet to an external destination, the IPv4 packet is encapsulated in an IPv6 packet for transport into the provider network. At the CGNAT (Carrier Grade NAT), the packet is decapsulated to IPv4 and NAT44 (which translates an IPv4 address to another IPv4 address) before delivering to the public Internet. This tunnelling of IPv4 packets enables IPv4 applications and IPv4 hosts to communicate with the IPv4 Internet over the IPv6-only links. Using this approach, a service provider can deploy IPv6 and still provide an IPv4 service.

A DS-lite CPE (Customer Premises Equipment) performs the IPv4-in-IPv6 encapsulation of the IPv4 packet sent by the device, setting the destination address of the IPv6 packet to the address of the DS-lite enabled CGNAT (Carrier Grade NAT), also known as the Address family Transition Router, AFTR.

A DS-lite CGNAT (Carrier Grade NAT) must adapt its NAT binding table. The source address of the encapsulating IPv6 packet (the address of the customer end of the IPv6 link) is added to the bindings beside the IPv4 source address and port. Because the IPv6 address is unique to each customer, the combination of the IPv6 source address with the IPv4 source address and port makes the mapping unambiguous. When a responding IPv4 packet is received from the outside, its IPv4 destination address and port can be correctly matched to a specific customer behind the NAT based on the IPv6 address in the mapping table. The packet IPv4 destination address and port can then be mapped to the inside IPv4 destination address and port, encapsulated in IPv6 using the mapped IPv6 address as the IPv6 destination address, and then forwarded to the customer. The mapped IPv6 address not only disambiguates the customer RFC1918 [9] address, it provides the reference for the tunnel endpoint.

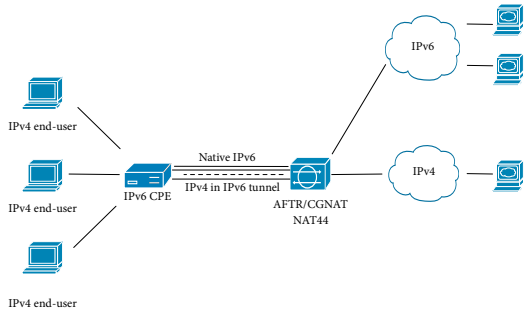

Fig. 1. Multiple Device Dual-stack Lite Topology

If a device sends an IPv6 packet, the packet is routed normally to the IPv6 destination. If a device sends an IPv4 packet, the CPE gateway performs the IPv4-in-IPv6 encapsulation, setting the destination address of the IPv6 packet to the address of the DS-lite enabled CGNAT (Carrier Grade NAT), also known as AFTR as shown in Fig. 1. This model allows use of Dual-stacked, IPv4-only, and IPv6-only devices behind the gateway.

A variation on the DS-lite Model, as depicted in Fig. 2 implements DS-lite on an individual end system rather than on a CPE device. The device is dual stacked, and therefore can send and receive both IPv4 and IPv6 packets. This model is

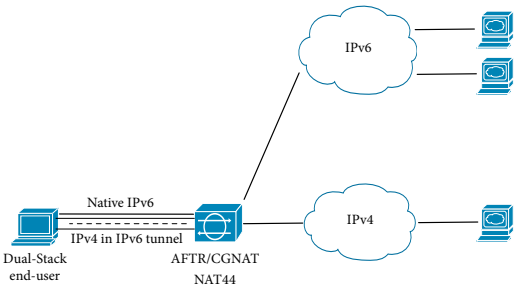

Fig. 2. Single Device Dual-stack Lite Topology

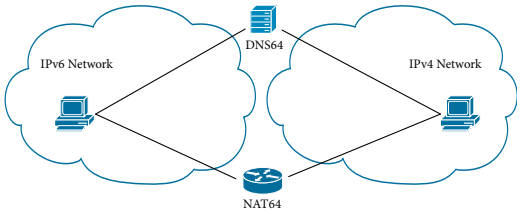

Fig. 3. `NAT64/DNS64` Topology

not only relevant to customers who connect a single PC, game system, or laptop to the Internet directly without a router, but it also has a great potential for mobile broadband.

One critical requirement for DS-lite is that its tunneling function must be added to existing customers CPE either through a software upgrade or by replacing the unit. This can be done when IPv6 functionality is added to the CPE.

B. NAT64

IPv4-to-IPv6 Network Address Translator (NAT64) [10] allows IPv6-only clients to contact IPv4 servers using unicast UDP, TCP, or ICMP. The headers of packets passing between an IPv6-only end system and an IPv4-only end system are converted from one protocol to the other using the IP/ICMP Translation Algorithm [11], allowing the end systems to communicate without knowing that the remote system is using a different IP version.

A special DNS Application Level Gateway (ALG), known as DNS64 [12], is used to trick IPv6 hosts into thinking that the IPv4 destination is an IPv6 address. The IPv6 host thinks that it is communicating with another IPv6 system, and the IPv4 system thinks that it is talking to another IPv4 system. Neither end system participates directly in the translation process.

The NAT64 server is the endpoint for at least one public IPv4 address and an IPv6 network segment of 32-bits (`64:FF9B::/96`). The IPv6 client embeds the IPv4 address it wishes to communicate with using these bits, and sends its packets to the resulting address. The NAT64 server then creates a NAT-mapping between the IPv6 and the IPv4 address, allowing them to communicate.

DNS64 describes a DNS server that, when asked for a domain's AAAA records, but only finds A records of the requested domain, synthesizes the AAAA records from A records. The first part of the synthesized IPv6 address points to an IPv6/IPv4 translator and second part embeds the IPv4 address from the A record. The translator in question is usually a NAT64 server as shown in Fig. 3.

The most significant limitation of this architecture is that all hosts and all applications within the NAT64 domain must be converted to IPv6. Legacy IPv4 hosts or IPv4 applications running on an IPv4 host will not work in this architecture. In addition the translation only works for cases where DNS is used to find the remote host address, if IPv4 literals are used the DNS64 server will never be involved. DNSSEC validation will also fail since DNS64 server needs to return records not specified by the domain owner.

III. NetFlow and F

NetFlow protocol exists in several different versions. The currently most widely deployed version is NetFlow version 5 [13]. The traditional definition of a flow uses a 7-tuple key. Specifically, a flow is a unidirectional sequence of packets sharing all of the following seven values:

1) Source IPv4 address
2) Destination IPv4 address
3) Source port for UDP or TCP, 0 for other protocols
4) Destination port for UDP or TCP, type and code for ICMP, or 0 for other protocols
5) IPv4 protocol field (indicating UDP, TCP, ICMP, etc.)
6) Ingress interface (identified by its interface index)
7) IPv4 Type of Service field (ToS)

If two IP packets differ in one of the seven fields, then they belong to different flows. However, many NetFlow analysis tools use a 5-tuple flow definition which includes the first five fields listed above to identify a flow. In addition to the key fields, a flow contains other accounting fields which may differ slightly depending on the NetFlow version. According to Cisco's NetFlow version 5 [13], these fields include the start and end times of a flow, the number of packets and octets in a flow, the source and destination Autonomous System (AS) numbers, the input and output interface numbers for the device where the NetFlow record was created, the source and destination network masks and, for flows of TCP traffic - a logical OR of all of the TCP header flags seen (except for the ACK flag). In the case of Internet Control Message Protocol (ICMP) traffic, the ICMP type and subtype are recorded in the destination port field of the NetFlow records.

A NetFlow record is created when traffic is first seen by a router or switch that is configured for NetFlow services. The record ends and is exported to the collector when one of the following conditions are met:

- The exporter detected the end of a flow. For flows representing TCP traffic, this would be the occurrence of a FIN or RST flag.
- The flow has been inactive for a certain period of time (Cisco default is 15 seconds but some operators are known to use much smaller values, such as one second).

- For long-lasting flows, the exporter should export the flow records on a regular basis (the Cisco default is 30 minutes after the start of the flow, but many installations use much shorter intervals, such as 5 minutes).
- The exporter experiences internal constraints, i.e., the internal flow cache fills and records must be exported due to memory constraints.

NetFlow version 5 records use a fixed format and are 48 bytes long. A NetFlow message can carry between 1 and 30 NetFlow records. The flow records in a NetFlow message are prefixed with a 24 byte NetFlow header carrying version information and general information about the exporter.

Cisco NetFlow version 9 [14] introduces a new template format, which includes all of the NetFlow version 5 fields and optionally includes extra information, such as Multiprotocol Label Switching (MPLS) labels and IP version 6 addresses and port numbers. The template format provides flexibility in the data format and thus enables router vendors to customize the data shipped in NetFlow messages.

There are two other compatible flow record export protocols: IPFIX and InMon's sFlow. The IPFIX (IP Flow eXport Information) working group of the IETF has chosen NetFlow version 9 as a base for developing a common, universal standard for exporting IP flow information from network devices. The IPFIX protocol [15] aims at being the standard version of NetFlow. The sFlow technology [16] provides mechanisms to sample traffic statistics from devices running sFlow agents. The main target of sFlow are high-speed switches that achieve data rates counted in 10s or 100s of Gbps.

F is a result of an evolution of the initial prototype language *Flowy* written in Python [7]. F is implemented in C, however, it is not simply a "re-implementation" in a different language, but a more efficient approach to flow record processing internally. Query execution speed has improved by a factor, and is now compatible to the existing flow processing tools that operate on absolute filters. Next, we describe the structure of the flow query language, look at how a query is defined and the complexity associated with relative filtering of flow records. F operates on NetFlow formatted data, and takes a query as an input. A typical query consists of several branches. Each branch contains a number of stages that perform their specific tasks based on the specified rules. The query also specifies which NetFlow trace file should be supplied as the input and where to store the output.

In order to understand the inners of the flow query language, we provide a flow diagram in Fig. 4, followed by a description of each stage. Starting in the left-most side of the diagram, F takes flow record data in a `flow-tools` [17] compatible format as an input. These flow records are read into memory. The rationale behind this decision is the random access pattern induced by every stage except for the filter stage. Current filtering tools (i.e., `nfdump` [18] or `flow-tools`) only apply absolute comparison of flow records and hence are able to read flow records and discard them "on the fly", without having to read the whole input data into memory first. Due to the random access pattern of relative comparisons (where a flow record can be compared to another flow record), the flow records have to reside in memory to not decrease performance drastically. Each record is a sequence of bytes in memory with the offsets and lengths of the enclosed fields stored in a special structure. Thus, to access the data, one has to first look up the required fields offset in this structure. Offsets and lengths of this kind are part of the flow query definition and define the location and size of the values that are to be compared later on in the filtering and grouping stages. The `splitter` stage distributes exactly the same copy of flow record location in the memory. This way, each of the subsequent branches have an equivalent access to the data, and can perform necessary operations on the data independently of each other. Each branch has three distinct stages: `filter`, `grouper` and `group-filter`. The functionality of the filter is very similar to that of the existing flow processing tools, and performs absolute filtering on different flow record fields, versus some constant values, i.e., port numbers or IP addresses. For instance, the filter stage can select all the flow records that have port 443 as the source port. The filter rules can also check for a combination of matching fields in a record, like: "select all flow records that have a source port of 80, SYN flag enabled and a destination address of www.xxx.yyy.zzz". The selected set of flow records that passed through the filtering stage move on into the grouper stage. The grouper forms groups of flow records according to certain rules. A typical rule can be: "group all flow records with the same source and destination IP addresses, as long as the start time of one record does not exceed the end time of another record by more than 500 milliseconds". The start and end times are taken from the start and end timestamps contained within flow records. These, newly-formed groups of flow records can also have meta-information about the contained flows. This is possible due to the aggregation mechanism defined as part of the query definition. A user can ask the grouper to provide information about the total number of bytes carried by all flow records in the group, the set of all source and/or destination addresses, total number of flow records contained or even return a bitwise OR on all of the flags contained in flow records. This meta-information can be useful at the next stages, where groups of flow records are filtered against fixed values, or when groups are evaluated against each other. The formed groups of flow records proceed to the group-filter, where an absolute filtering occurs, and those groups that don't satisfy certain filtering rules are discarded. A typical filtering rule at this stage can check whether there are at least n flow records in the group or if a group has certain flags enabled. The filtered groups proceed to the merger. The merger defines which branches (i.e., A, B, etc.) should it take groups from and which operations should it perform on them. A simple example could be matching source IP address of groups in branch A to the destination IP address of groups in branch B: $A.srcip = B.dstip$. One could also check if the groups from branch A are smaller than their counterparts from branch B. At this stage one could define rules based on Allen interval algebra in order to find matches where a group from one

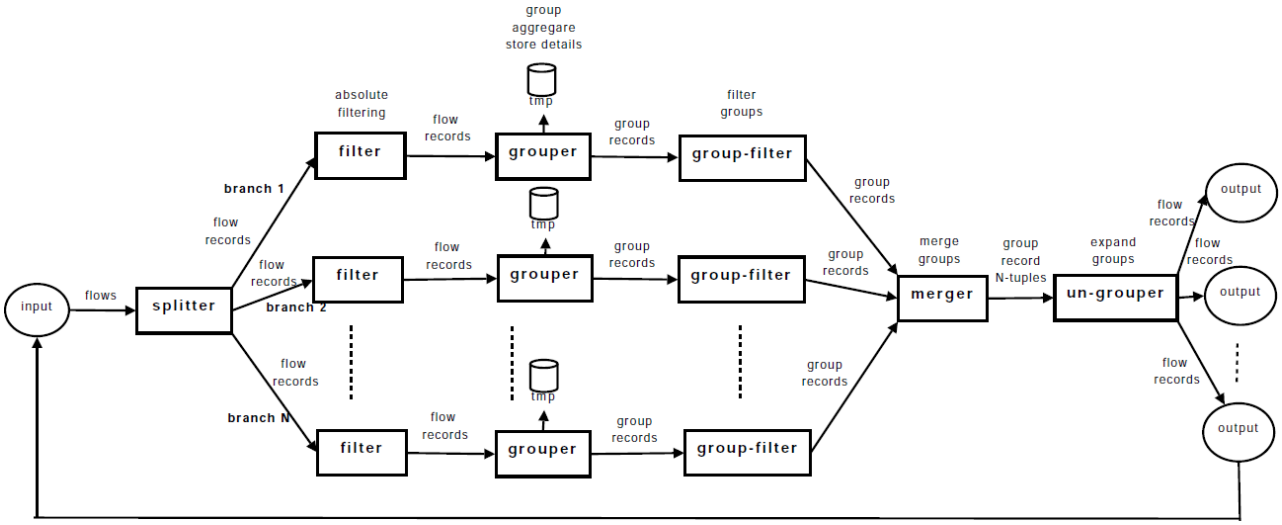

Fig. 4. F's flow diagram

branch should occur during the occurrence of a group from another branch. Besides, one could also check for situations when a group from branch A starts before a group from branch B, and ends before, during or after the end time of a group from branch B. A curious reader is suggested to consult the original paper [19] by J. F. Allen for a full set of interval algebra relations. Once all of the stages are traversed, the groups are taken to the `ungrouper` stage, where they are unbundled to produce individual flow records as the output.

IV. Application Operation Under DS-lite

We used three physical machines in our Dual-stack setup. One of the Debian machine works as our Customer Premise Equipment (CPE) [8], one as our Carrier Grade NAT (CG-NAT) [8] and the third one is our IPv4 only host running Mac OS X as shown in Fig. 5.

The CGNAT machine is a combination of an IPv4-in-IPv6 tunnel end-point and an IPv4-IPv4 NAT implemented on the same node [8].

The machine has two interfaces, one (`eth0`) with a Global Scope IPv6 address (`2020::1`) to underpin the virtual IPv4-in-IPv6 tunnel and the other one (`eth1`) with a public IPv4 address to reach the IPv4 world. A virtual interface (`tun0`) was defined with an IPv4 address (`172.0.0.1`) on top of `eth0` to establish an `ipip6` tunnel to the CPE. A kernel module `ip6_tunnel` was required to be loaded to get the `ipip6` tunnel working. At this point all `172.0.0.0/64` traffic was routing to `tun0` and all `2020::/64` traffic was routing to `eth0`. We used `iptables` to forward all the traffic from `tun0` to `eth1` and to allow NATing with the public IPv4 address. We also enabled IPv4 forwarding on all interfaces for the `iptables` forwarding to work.

The CPE machine is a layer 3 device in the customer premise functioning as a home gateway that is connected to the service provider network. The B4 element implemented on a Dual-stack capable node within the CPE creates a tunnel to route all IPv4 requests to the CGNAT machine [8].

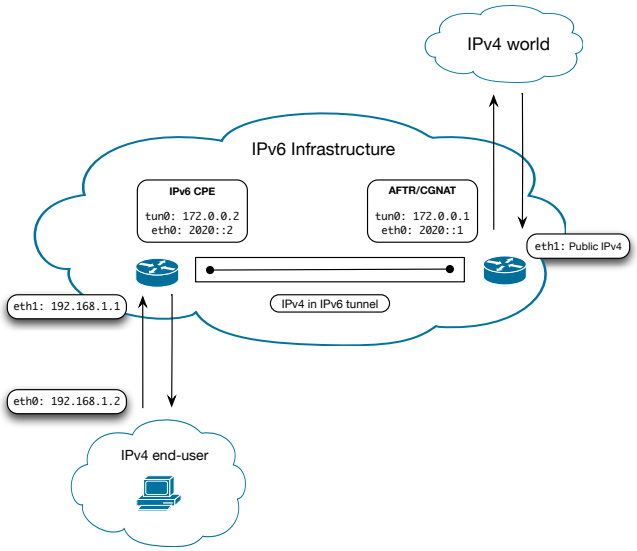

Fig. 5. Dual-stack lite setup

The machine has two interfaces, one (`eth0`) with a Global Scope IPv6 address (`2020::2`) to underpin the virtual IPv4-in-IPv6 tunnel similar to the CGNAT machine, and the other one (`eth1`) with a private [9] address (`192.168.1.1`) which serves as the DHCP server and a gateway point for IPv4 clients directly connected to it. We added a default route for all IPv6 packets to pass through `eth0` with `2020::1` (CGNAT) as their next hop. At this point we could test IPv6 connectivity from both endpoints.

A virtual interface (`tun0`) was defined with an IPv4 address (`172.0.0.2`) on top of `eth0` to establish an `ipip6` tunnel to the CPE. A kernel module `ip6_tunnel` was required to be loaded to get the `ipip6` tunnel working. We added a default route for all IPv4 packets to pass through this virtual tunnel.At

this point we could test IPv4 connectivity of tunnel endpoints.

We also added a specific route for our private subnet `192.168.0.0/16` to route all local traffic through to `eth1`. We again used `iptables` to forward all the traffic, but this time from `eth1` to `tun0` and to allow NATing with the tunnel's IPv4 address (`172.0.0.2`). We also enabled IPv4 forwarding on all interfaces for the `iptables` forwarding to work. The IPv4 only host has one interface `eth0` with a private [9] address. A default IPv4 route was added on this machine to send all its IPv4 requests to the CPE. At this point we could test its IPv4 connectivity with the IPv4 world.

Once the setup of DS-lite was performed, we carried out a number of application evaluations. As such, we ran the following set of applications and protocols.

- Safari 5, Google Chrome 10, Firefox 4, Opera 11
  - -- WebMail: Gmail using TLSv1
  - -- Media: YouTube using Flash and HTML5
  - -- Google Maps
  - -- Web Chat: Gmail, Yahoo, Freenode IRC
  - -- HTTP Downloading
- Apple Mail 4
  - -- IMAP: Gmail and University Exchange
  - -- POP3: Gmail
  - -- SMTP: Gmail and University Exchange
- Instant Messaging and VoIP
  - -- iChat, Skype
- Miscellaneous
  - -- SSH, FTP, IRC, Git, Mercurial, OpenVPN, Transmission: bit torrent

Despite our expectations for some sort of failures, we were not able to identify any among the ones we listed above. All of these applications worked with an installation of the transition technology "out-of-the-box". This result it itself demonstrated that this transition technology was already suitable to fit the scope of its application and deliver a great performance. Since there were no failures, we naturally could not define any *failure signatures* in F. Next, we proceeded to evaluate NAT64 on the set of the same applications, and the results were different, as presented in the next section.

V. Application Operation Under NAT64

A Debian machine functions as our IPv6 only host. It has one interface (`eth0`) with a private IPv4 address [9] to send out DNS queries, one global scope IPv6 address to directly reach the IPv6 world, and another global scope IPv6 address to reach the IPv4 world using NAT64. Another Debian machine functions as our DNS64 box to which our IPv6 only host sends out all DNS queries and is within the same subnet as the IPv6 only host. To achieve this, our IPv6 only host defines the DNS64 box as the only nameserver in `/etc/resolv.conf`

In order to catch all IPv6 DNS requests, `totd` [20] was installed on the DNS64 box which is a stateless user-space implementation of DNS64 which forwards the requests to a real IPv4 nameserver and then constructs fake IPv6 address based on a pre-defined IPv6 NAT64 prefix (`64:FF9B::`) and IPv4 address it finds from the real nameserver for the DNS request.

At this point we were able to receive fake IPv6 addresses for DNS requests for a domain name from our IPv6 only machine.

The testbed that runs the NAT64 module is a physical machine running Debian with 2 interfaces. One of the interfaces (`eth0`) has a public IPv4 address and is connected to the IPv4 world. The other interface (`eth1`) has a Global Scope IPv6 address to connect back to our IPv6 only host. In order to route all IPv6 requests with a NAT64 prefix (`64:FF9B::`) to the NAT64 machine, a default route was setup on the IPv6 only machine. The NAT64/DNS64 experimental setup is shown in Fig. 6.

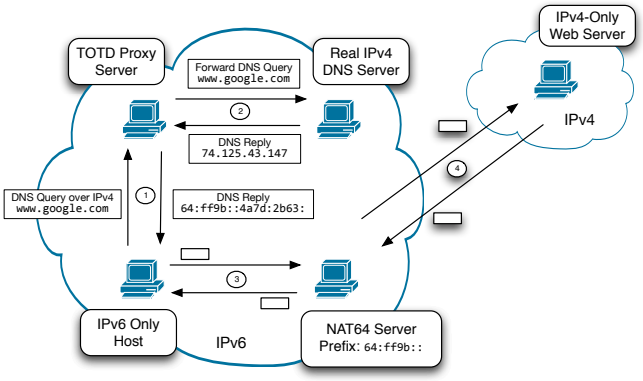

Fig. 6. NAT64/DNS64 Setup

Finally, in order to get the IPv6-IPv4 translation working, we installed `Ecdysis` [21] (on the NAT64 machine) which is an open-source implementation of NAT64 to translate all IPv6 requests received at `eth1` (from the IPv6 only machine) to IPv4 requests and sends it to `eth0` which are then routed to the IPv4 world. At this point we could test connection to IPv4 content from our IPv6 only host. Since, our IPv6 only host is also directly connected to IPv6 world, we we were also able to reach IPv6 only websites.

We tested the same set of applications and protocols as presented in section IV, and for majority of applications the experiences have been similar to those of DS-lite. Namely, most applications functioned well with the initial setup. A setup that did not require any fine-tuning. However three of the applications failed to inter-operate with NAT64: Skype, OpenVPN and Transmission: bit torrent. In the next section we discuss the reasons for failure of these applications and demonstrate how to define the failure signature of Skype in F.

VI. Failure Analysis and Signature Definition

These failures correspond to similar experiences with NAT64, documented in two other independent studies [4] [5]. Once a failure was detected, the flow record trace was inspected to identify the sequence of flow record exchanges representing a "failure signature". The detected failure signatures were defined as a query in F. These failure signatures

could now be used to scan flow record traces in order to automatically identify potential application- or protocol-related problems that different users or network endpoints are experiencing. Next, we will go over the sequence of flow-record exchanges that represent Skype's failure, and convert it into F's flow query. This query can further be used to scan through other NetFlow traces in order to identify similar problems.

There is a particular sequence of events that occur at the start up and a sign-in procedure of Skype. At the start of the sign in procedure Skype tries to discover clients in the local network. It tries to send a multicast message using the multi-cast DNS (mDNS) protocol. To perform that, it uses a specific destination IP address-port combination (224.0.0.251:5353). This message forms a single flow and has a brief duration (we tested it only on a single network setup; however a typical value is around 3 seconds). Once this procedure is over (and unsuccessful, in case of the NAT64 setup) Skype proceeds to establish a contact to the Skype login server via the HTTPS protocol (port 443). The login server resides on an IPv4 address, however it is converted into the IPv6 format by the DNS64 server. There are three subsequent attempts to contact the login server and each of the attempts represents a flow record of a longer duration (experiments have shown that this duration is usually 57 seconds, however in a query the threshold is set with a +/-2 seconds "buffer" zone). However, each of the subsequent flows always had a decreasing number of packets. Also, each of these flow records is targeted at the same IPv6 address. The source port number increases by 1 for each subsequent flow (e.g., 2001:63..:708::2.43148, 2001:63..:708::2.43149 and 2001:63..:708::2.43150). The login session fails after three unsuccessful attempts, and the user is prompted to enter her/his password again.

Let's dissect and analyse a query representing this signature in F. The simplest approach is to define four separate branches. One branch devoted for the initial mDNS request for Skype clients on the local network, while the other three are for subsequent attempts to connect to the login server. This approach would not be the most optimal one (remember that the number of branches has the highest "weight" in algorithmic complexity of F), but it is rather simple to follow. In mDNS branch we need to filter for those records that have source and destination ports set to 5353 and the destination address would be a known multicast address of 224.0.0.251. The duration of this flow is typically around 3 seconds, but we can be more lenient and provide a buffer of two seconds (at a price tag of getting extra/unrepresentative flow records).

```
3 filter f-mDNS {
4    dstport = 5353
5    srcport = 5353
6    dstip = 224.0.0.251
7    duration > 1 sec
8    duration < 5 sec
9 }
```

The sequence of three flow records have a very simple filter, which scans for flow records with 443 as their destination port and that last from 55 to 59 seconds (a 2-second buffer).

```
11 filter f-login1 {
12    dstport = 443
13    duration > 55 sec
14    duration < 59 sec
15 }
```

Once, each of the branches select matching flow records, we need to form groups from them. Since we have four branches, each branch has its own task, therefore the groupers are also rather simple. A general F query rule is that "the number of branches is inversely proportional to the number of rules at each of the stages". That is, the more branches there are, the simpler is the logic (less rules) in each of the branch's stages. Since mDNS branch is represented by a single flow-record, the grouper of mDNS branch essentially contains meta-data. The groupers of three subsequent Skype login server accesses should also have a single flow record associated with them. The source ports should not be different. We also know that the number of packets associated with each flow decreased. Therefore, we need to count the number of packets in each branch (or a single flow record). This is done by using the `aggregate` keyword to store the sum into `pckt-sum`. It should carry information about the source port as they should vary by one among three branches. It is important to note time durations of each group, since we will compare it to the duration from `mDNS` branch. These three fields will be used at the merger stage. As the last step we need to count the number of flow records in each group, as we will filter out those groups that have more than one record at the `groupfilter` stage.

```
38 grouper g-login1 {
39    module g1 {
40       srcport = srcport
41       dstip = dstip
42       dstport = dstport
43    }
44    aggregate srcip, dstip,
45       sum(packets) as pckt-sum,
46       srcport, td,
47       count(rec_id) as n
48 }
```

The `grouper` stage is similar for the other two branches. The task of groupfilters on all four branches is to verify that each group contains a single flow. Therefore they will check if n from `grouper` stage is equal to one. The `merger` stage will join all four branches. The merger should check if the group from `mDNS` branch occurred and completed before any of the other three branches (ll. 108-110). Source and destination IPs from `LOGIN` branches should match (ll. 93-96). Besides, it checks whether the source ports for the non-`mDNS` joining branches have a difference value of one (ll. 98-99). The number of packets in each subsequent `LOGIN` branch was also decreasing (ll. 101-102). The duration of mDNS flow is also much smaller than durations from three other branches (ll. 104-106).

```
90 merger M {
91    branches mDNS, LOGIN1, LOGIN2, LOGIN3
92
93    LOGIN1.srcip = LOGIN2.srcip
94    LOGIN2.srcip = LOGIN3.srcip
95    LOGIN1.dstip = LOGIN2.dstip
96    LOGIN2.dstip = LOGIN3.dstip
97
```

```
 98    LOGIN1.srcport = LOGIN2.srcport rdelta 1
 99    LOGIN2.srcport = LOGIN3.srcport rdelta 1
100
101    LOGIN1.pckt-sum > LOGIN2.pckt-sum
102    LOGIN2.pckt-sum > LOGIN3.pckt-sum
103
104    mDNS.td < LOGIN1.td
105    mDNS.td < LOGIN2.td
106    mDNS.td < LOGIN3.td
107
108    mDNS < LOGIN1
109    mDNS < LOGIN2
110    mDNS < LOGIN3
111 }
```

As the last step we interconnect all branches, supply input trace and receive output following the `ungrouper` stage (due to space constraints: `br` stands for `branch`).

```
115 "input"-> S
116 S br mDNS  ->f-mDNS  ->g-mDNS  ->gf-mDNS  ->M
117 S br LOGIN1->f-login1->g-login1->gf-login1->M
118 S br LOGIN2->f-login2->g-login2->gf-login2->M
119 S br LOGIN3->f-login3->g-login3->gf-login3->M
120 M -> U -> "output"
```

The more optimized way of defining this query would have been by forming groups that don't have a condition for having the same source port, and then admitting only those groups that have exactly 3 flow records through the `groupfilter`. Not having a condition for matching source ports in the grouper would imply the additional `delta 1` condition in the `groupfilter`, which implies that a group contains flow records that match all of other `grouper`'s rules, plus have an increasing (by one) sequence of source ports.

A significant limitation of the NAT64 architecture is that all hosts and all applications within the NAT64 domain must be converted to IPv6. Legacy IPv4 hosts or IPv4 applications running on an IPv4 host will not work under this architecture [22]. In addition the translation only works for cases where DNS is used to find the remote host address, if IPv4 literals are used the DNS64 server will never be involved. The latter is essentially the reason for failing of the other two applications (OpenVPN and Transmission). DNSSEC validation will also fail since DNS64 server needs to return records not specified by the domain owner.

So far, we talked about applications that require user's interaction, however there can be scenarios where an application is assumed to be running in the background and processing incoming and outgoing traffic streams, while in reality it can be experiencing a faulty state, and go unnoticed for longer time durations. These problems may occur due to misconfiguration or due to rapid changes on the network (especially could be applicable to the OpenFlow protocol [23], as its operation is tightly integrated with flow records). ISPs could also detect certain failures associated with user's overall experiences and take appropriate measure in adjusting or tailoring their services in order to eliminate these failures. As such, failure detection by NetFlow failure signatures is not limited just to the IPv6-to-IPv4 transitioning mechanisms. It can be applied to identify failures in other network scenarios or in general traffic. While F, with its stream-oriented query definition, is a potential solution for defining these queries.

As we have already established in [24], there can be false positives associated with such methodology. Usually false positives occur in cases when the signatures were defined not rigorously enough or in cases when different applications have very similar flow record signatures (e.g., we experienced a strong similarity in iTunes and Amarok music players' signatures).

VII. Related IPv6 Transition Technologies

There are also some different implementations of NAT64/DNS64. TAYGA [25] is a user space stateless NAT64 implementation for Linux that uses the `tun` driver to exchange IPv4 and IPv6 packets with the kernel. It is intended to provide production-quality NAT64 service for networks where dedicated NAT64 hardware would be overkill. Routing TAYGA's IPv4 path through a stateful NAT44 can also make it stateful.

Address Plus Port [A+P] [26] uses a NAT in (or close to) the customer premise for access to the IPv4 Internet. The Service Provider conveys both an IPv4 address (as done today) and a range of TCP/UDP ports to the NAT. Outgoing IPv4 traffic is NATted to that range of TCP/UDP ports, and the Service Provider routes packets to the appropriate customer using both the destination IPv4 address (as done today) and destination TCP/UDP port.

Teredo [27] is a technology to provide IPv6 connectivity to IPv4 endpoints whose service providers haven't deployed IPv6 yet. Teredo is a platform-independent tunnelling protocol designed to provide IPv6 connectivity by encapsulating IPv6 packets within IPv4 User Datagram Protocol (UDP) giving datagrams that can be through NAT devices and on the IPv4 Internet. Other Teredo nodes elsewhere called Teredo relays that have access to the IPv6 network then receive the packets and route them on.

NAT444 [28] is very similar to Dual-Stack Lite, however it has an advantage of not imposing any special requirement on the CPE. It uses three layers of addressing instead of an IPv4-in-IPv6 tunnel. One layer uses a private IPv4 block behind the CPE, another uses a separate private IPv4 block between the CPE and CGNAT and the third layer uses a public IPv4 address routing outside from the CGNAT to the IPv4 world.

VIII. Conclusion

In accordance with the goals of this study, we evaluated the operation of a variety of applications and protocols under two different IPv4-to-IPv6 transition mechanisms. All of the tested applications performed well with the "out-of-the-box" setup of DS-lite, while some problems occurred with NAT64 inter-operation. Namely, we identified three applications that failed to function with NAT64 setup. Once these failures were detected, we carried out an investigation to identify a sequence of flow record exchanges in order to define a stream-oriented query in F. The failure signature of Skype was studied in great detail and can now be used to automatically detect similar failures in a general set of NetFlow traces. Using such signatures, an interested party can scan its NetFlow trace files in order to automatically identify failures associated

with a certain IPv6 transition mechanism, their frequency and associate end-points (hosts) that experience similar problems. While a larger, and more exhaustive study needs to be carried out for a number of different failure signatures and various real-life IP flow record traces, we have still managed to set the baseline for this type of approach to automatic failure identification under IPv6 transition mechanisms.

References

[1] IPv6 Adoption Monitor. [Online; accessed 14-March-2011].

[2] APNIC IPv4 Address Pool Reaches Final /8. [Online; accessed 20-May-2011].

[3] Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied. [Online; accessed 14-March-2011].

[4] J. Arkko and A. Keranen. Experiences from an IPv6-Only Network. Internet-Draft (Informational), October 2011.

[5] H. Hazeyama and Y. Ueno. Experiences from an IPv6-Only Network in the WIDE Camp Autumn 2011. Internet-Draft (Informational), October 2011.

[6] V. Marinov and J. Schönwälder. Design of a Stream-based IP Flow Record Query Language. In *Proc. DSOM 2009*, number 5841 in LNCS, pages 15--28, Venice, October 2009. Springer.

[7] Kaloyan Kanev, Nikolay Melnikov, and Jürgen Schönwälder. Implementation of a Stream-based IP Flow Record Query Language. In *Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security*, AIMS'10, pages 147--158, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] A. Durand, R. Droms, J. Woodyatt, and Y. Lee. Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion. RFC 6333 (Proposed Standard), August 2011.

[9] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[10] M. Bagnulo, P. Matthews, and I. van Beijnum. Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. RFC 6146 (Proposed Standard), April 2011.

[11] X. Li, C. Bao, and F. Baker. IP/ICMP Translation Algorithm. RFC 6145 (Proposed Standard), April 2011.

[12] M. Bagnulo, A. Sullivan, P. Matthews, and I. van Beijnum. DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers. RFC 6147 (Proposed Standard), April 2011.

[13] Cisco Systems. NetFlow Services Solution Guide, January 2007.

[14] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[15] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[16] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational), September 2001.

[17] S. Romig. The OSU Flow-tools Package and CISCO NetFlow Logs. In *Proceedings of LISA '00*, pages 291--304, Berkeley, CA, USA, 2000. USENIX Association.

[18] P. Haag. Netflow Tools NfSen and NFDUMP. In *Proc. 18th Annual FIRST Conference*, June 2006.

[19] James F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26:832--843, November 1983.

[20] The totd ('Trick Or Treat Daemon') DNS proxy. [Online; accessed 14-March-2011].

[21] Ecdysis: open-source implementation of a NAT64 gateway. [Online; accessed 14-March-2011].

[22] Tools and Strategies for Coping with IPv4 Address Depletion: Technologies for an IPv4 Address Exhausted World. White paper, Juniper Networks, 2010.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38:69--74, March 2008.

[24] Vladislav Perelman, Nikolay Melnikov, and Jürgen Schönwälder. Flow Signatures of Popular Applications. In *12th IFIP/IEEE International Symposium on Integrated Network Management*, 2011.

[25] TAYGA, Simple, No-fuss NAT64 for Linux. [Online; accessed 14-March-2011].

[26] R. Bush. The Address plus Port (A+P) Approach to the IPv4 Address Shortage. RFC 6346 (Experimental), August 2011.

[27] C. Huitema. Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs). RFC 4380 (Proposed Standard), February 2006. Updated by RFCs 5991, 6081.

[28] I. Yamagata, Y. Shirasaki, A. Nakagawa, J. Yamaguchi, and H. Ashida. NAT444. Internet-Draft (Informational), January 2012.