IPv6 Address Obfuscation by Intermediate Middlebox in Coordination with Connected Devices

Florent Fourcot[1,2], Laurent Toutain[1], Stefan Köpsell[2], Frédéric Cuppens[1], and Nora Cuppens-Boulahia[1]

[1] Institut Mines-Télécom; Télécom Bretagne
{first.last}@telecom-bretagne.eu
[2] TU Dresden; Faculty of Computer Science
stefan.koepsell@tu-dresden.de

Abstract. Privacy is a major concern on the current Internet, but transport mechanisms like IPv4 and more specifically IPv6 do not offer the necessary protection to users. However, the IPv6 address size allows designing privacy mechanisms impossible in IPv4. Nevertheless existing solutions like Privacy Extensions [20] are not optimal, still only one address is in use for several communications over time. And it does not offer control of the network by the administrator (end devices use randomly generated addresses). Our IPv6 privacy proposal uses ephemeral addresses outside the trusted network but stable addresses inside the local network, allowing the control of the local network security by the administrator. Our solution is based on new opportunities of IPv6: a large address space and a new flow label field. In combination with Cryptographically Generated Addresses, we can provide protection against spoofing on the local network and enhanced privacy for Internet communication.

Keywords: IPv6, Privacy, Security, Address Management.

1 Introduction

If IPv4 is still the most popular IP stack, IPv6 leaves the laboratory to be deployed on the Internet. For example, a lot of popular websites activated IPv6 on 6th June 2012 [1], and the French Internet provider "Free" offers IPv6 connectivity for new clients by default. This activation of IPv6 is not without privacy issues like the possibility to trace a device, thanks to the interface identifier stability [19]. Indeed, an IPv6 address is made of two parts. The first one is the routing information, read by routers across the Internet. The second part is the interface identifier, locally generated but worldwide readable. The default stateless configuration uses the MAC address to generate this interface identifier, without other parameters [17]. This means that initially the interface identifier of a device is always the same, regardless of the connected network; the device is traceable across the whole world.

T. Bauschert (Ed.): EUNICE 2013, LNCS 8115, pp. 148–160, 2013.

To obfuscate this interface identifier, there exist full fledge anonymous communication solutions, e.g. based on the ideas of mixes [11] like AN.ON [9] or Tor [13]; DC networks [10] and many similar proposals. This class of solutions offers high grade of protection even against powerful attackers but at the price of complex design and deployment. In our paper we focus on a light weight construction which offers protection only against weaker attackers. More specific we assume an attacker outside of the trusted network. A prominent example would be a web service which tries to reidentify its users.

One existing light weight solution called Privacy Extensions is defined in RFC 4941 [20]. Privacy extensions are compatible with the stateless autoconfiguration, but this solution is not really satisfactory. Usually the same address is used over a long period of time, because changing the address implies disconnection of all active connections. Thus there is a trade off between relatively stable connections versus a real privacy gain. Second, by using only one address, Privacy Extensions do not take advantage of the large IPv6 space. All applications of the connected device will use the same address, therefore an eavesdropper can easily link different communications of the same device, e.g. Instant messaging communication and the Web traffic.

From a network administrator perspective, Privacy Extensions might be unacceptable because it makes logging user's connections more cumbersome. This logging is mandated by law in some countries, i.e. administrators have to reveal who was using a given address in case of court order or police investigations. One solution is to deploy the complex DHCPv6 protocol with distribution of temporary addresses, standardized in RFC 3315 [14]. With this stateful protocol, assignments of temporary addresses can be stored and are accessible to future requests. Nevertheless, it does not solve the disconnection problem in case of address switching and still does not utilize the large address space.

Another way to manage IPv6 addresses is to use Cryptographically Generated Addresses (CGA) [8]. CGA addresses are generated by the host itself, and can be used to improve the security of communications. One example is the use of CGA in conjunction with SEcure Neighbor Discovery [6] to prevent address spoofing on the local network. The interface identifier of a CGA address is based on a cryptographic hash function, and looks like a random address for an outsider. But the computational cost of CGA generation with an adequate security level is high [3] and prevents to use it as a privacy solution with high frequency of CGA calculation.

Our solution does not change the management of IPv6 assignment, and is compatible with stateless autoconfiguration, DHCPv6 and CGA. To improve the privacy of users, we introduce a middlebox, traditionally the border router of the network or the local firewall. On an IPv4 network, the middlebox is frequently a Network Address Translator (NAT) [22], and is already more "intelligent" than a simple router. An example of middlebox on the IPv6 network is a Network Prefix Translation (NPTv6) device [24]. NPTv6 is an experimental solution to change the prefix of addresses, using a one-to-one mapping. In contrast, our middlebox is in charge of spreading addresses across all locally available addresses, typi-

cally a /64 network, this means more than 10^{20} addresses. To accomplish this spreading, the middlebox assigns a random address for each *flow* sent by local devices and rewrites the source address accordingly. This spreading divides the network in one trusted space with stable addresses and one untrusted network (the Internet) with ephemeral addresses. Moreover, this rewriting can be easily activated or deactivated for each *flow*.

This paper is organized as follow: we describe the architecture of our solution in Section 2, we describe our implementation in Section 3, we discuss impacts and consequences in Section 4. We continue in Section 5 with a solution for assigning flow labels by application. Section 6 presents a performance measurement. Finally, Section 7 and 8 conclude the paper.

2 Proposal of Architecture

2.1 Overview of New IPv6 Opportunities

Our solution is based on two new opportunities offered by IPv6, namely a large address space and flow labels.

Large Address Space: IPv6 addresses are encoded in a 128 bits field, it means that 2^{128} addresses are available, more than 10^{28} addresses for each people alive today. As shown in the introduction, each address is split into two main parts and the first part is used for the routing and identifies a network. This main part of network identifier is assigned by an Internet provider to a company or home network. The second part of the address is the interface identifier. It can be locally managed and identifies the connected device on the network. The size of the locally manageable part is not always the same, but the recommendation is to give not less than 64 bits to the end network [12]. This means that with thousands of connected computers, each with one address in use, less than 0.00000000000001% of the address space is used. Therefore, if the IPv4 address management has to minimize the number of addresses in use, we can build new paradigms based on abundance of addresses in IPv6.

The flow label is a new 20 bits field in IPv6. The usage has been experimental for a long time, but after extensive discussions [4, 18], it has been standardized in 2011 [5]. The purpose of the field is to simplify the flow classification in order to apply some policies without complex packet inspection (like the well known 5-tuple in IPv4). The flow label is added by connected devices, which are easily able to discriminate flows without additional computational overhead. The recommended usage of this field is Quality of Service oriented (for example to prioritize real time communications), but this classification can have other use cases, summarized in RFC 6294 [18].

2.2 Overview of the Solution

To increase the user's privacy, we propose to assign one external address per flow. For each independent flow of packets, the connected device assigns a new flow label[1]. Then, the middlebox assigns a new external address to each pair of (*internal IP address*, *flow label*) and rewrites the source addresses of the outgoing packets and the destination addresses of the incoming packets (cf. Figure 1). Because the middlebox is in position of a border router, it receives all the packets from the local network. Therefore, it does not need to send extra neighbor discovery packets. In contrast, if the rewriting happens on the end devices, this solution implies some active neighbor discovery.

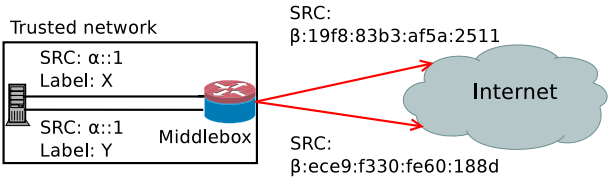

Fig. 1. Architecture of the solution: spreading by the middlebox

Since some applications can be incompatible with address rewriting (similar to the implications of NAT in IPv4 [16]), a flow label set to zero is a signal to forbid rewriting. This special label can be used if a temporary address is undesirable, for example in case of IP source address filtering on the destination device or an incompatible application layer.

To summarize, the intelligence to discriminate flows and optimize privacy is based on the end device, and all rewritings are based on the middlebox, i.e. under control of the network administrator. There is no need to change local address assignment policy. The middlebox should be located between the local firewall and the Internet; it prevents to rewrite firewall policies.

2.3 Computation on the Middlebox

Since the "intelligence" of flow classification is with in in the connected devices, the middlebox does not need to do a complex parsing of packets' headers, and to follow a TCP stream in a stateful way. But it has to maintain a context to perform rewriting (cf. Table 1). For each outgoing packet with an unknown pair (*internal IP address*, *flow label*) (short $(IP_{int}, label)$) the middlebox creates a context and generates a random interface identifier, which creates an address by concatenation with the prefix, named *external IP address* (short IP_{ext}). The stored context is a 3-tuple $(IP_{int}, label, IP_{ext})$, and all following packets matching the pair $(IP_{int}, label)$ will be rewritten with the IP_{ext}. For all incoming

[1] Given the 20 bits for a flow label, the risk of exhaustion is quite low.

packets, the middlebox rewrite the destination address with IP_{int} if a context exists, or applies standard routing and firewall policies.

Note that a flow (defined by all packets sharing the same source IP and the same flow label) can be made of more than one TCP connection (or other transport protocols). For example, we recommend to use the same flow for all the elements of a given Web page. The middlebox itself does not care about upper protocol layers, because the flow assignment is done on the end device.

Table 1. Example of context on the middlebox

Internal IP	Flow label	External IP
α::1/64	120137	α:ece9:f330:fe60:188d/64
α::1/64	4162	α:19f8:83b3:af5a:2511/64
α::2/64	647513	α:6c40:9951:605f:8e03/64

2.4 Flow Label Assignment by Application

The connected device is in the best place to discriminate flows and to assign flow labels. For example, a peer-to-peer application probably needs to use the same address for more than one TCP connection, a Web browser knows if one connection is related to another, etc. In our case, the best way is to patch the application to assign flow labels efficiently.

To help a fast deployment of our solution, we propose an assignment of a flow label per application in Section 5.2.

3 Implementation of the Middlebox

To test our solution, we implemented and deployed the middlebox in a real network environment. The middlebox is based on a standard Linux Kernel, and we added a Netfilter module to spread addresses. The middlebox has to rewrite addresses for outgoing and incoming packets. The limitation of our current implementation is that it can only be load on one interface connected to the Internet and does not yet support multihoming.

3.1 Packets Processing

Outgoing Packets: for each outgoing packet, we read the flow label information. If this label is zero, we stop the work of the module and the standard policy of the kernel is applied. Otherwise, we check if a context with the pair of source address and flow label already exists.

If we do not have a context, we have to create one. The first step is to generate a random address, and to check if the address is not in collision as explained in Section 4.2. The prefix part is static and can not be rewritten, but it is possible

to configure the length of the prefix (routing information), to maximize the size of the rewritable address part. We added the new external address to the pair (*source IP address*, *flow label*), and happen this context to the context table.

If a context exists, or after the initialization of a context, we rewrite the source address with the value stored in the fetched context. Afterwards we have to adjust the transport layer checksum. There is no standard way to rewrite this checksum, therefore we have to write code for each protocol. Currently, our implementation supports the three most popular protocols: TCP, UDP and ICMP (cf. Section 4.1). After this rewriting, we return the packet to apply standard kernel policy.

The flow label is no longer useful after the middlebox, and is set to zero.

Incoming Packets: for incoming packets, we only read the destination address. We check if a context exists for this address. If not, the packet is transmitted in the standard kernel way. Otherwise, we rewrite the destination address with the value stored in the context and we adjust the transport layer checksum.

3.2 Identification of a Context

The identification of a context has to be efficient on both directions. The identification of outgoing flows is done by matching the source address and the flow label with all existing contexts. For incoming packet, the identifier of the context is the destination address. We implement these searches with two hash tables, one for outgoing packets using "source IP address + label" as key value, and the second one for incoming packet using destination address as key value. This double hash table allows fast matching between packets and contexts.

3.3 Cleanup of Old Context

At the termination of a flow, the middlebox should remove the corresponding context to potentially reassign the address and to free the memory used. But in the IP network, there is no concept of connection and there are no communication messages to signal the end of a flow.

In IPv4 networks, RFC 4787 and 5382 [7, 15] give some recommendations to maintain a connection context for a NAT. But in your case, it is not possible (and desirable) to trace the state of a TCP connection. On the one hand, a flow can span more than a single TCP connection, on the other hand additional transport protocols can be in use. The only available solution is to introduce a timeout after an inactivity period. It should not be less than 120 seconds, according to recommendation for IPv4. A large timeout period will help to avoid breaking established connections, at cost of resource consumption. Based on our empirical tests, we recommend a value of 30 minutes, which give a good trade-off between resource consumption and connection stability. But the local administrator could overwrite this standard configuration in case of particular needs like long inactive TCP connections.

4 Impacts and Consequences

4.1 Checksum Computation

In IPv6, there is no checksum contained in the IP header but the transport layer protocols like TCP and UDP are in charge of error detections and therefore utilize a checksum. This checksum needs to be adapted if a rewriting happens. Fortunately, the flow label is not part of the checksum calculation and can be overwritten without implication. Nevertheless, the source address rewriting has an impact on the transport layer protocol checksum.

The large IPv6 address space supports some checksum neutral modifications, like in NPTv6 [24]. But in our case this solution is unacceptable. A checksum neutral modification gives a way to group all rewritten addresses of a device, with a simple checksum calculation of the source address. This removes the unlinkability between several random addresses.

But thanks to good properties of the standard Internet checksum, the cost of checksum computation is low, and an incremental update is possible [21]. We do not need to completly recompute new checksum C_{new} of a packet, since we can easily add the difference C_D between the 16-bit checksum C_{int} of the internal IP address and the 16-bit checksum C_{ext} of the external IP address to the already computed checksum C_{old}: $C_{new} = C_{old} + C_D$. During the initialization of the context, the middlebox will calculate $C_D = C_{int} - C_{ext}$ once, and caches this value.

4.2 Risk of Collision

Each generated address has to be unique behond the all local subnet, in order to not disturb the network. As a consequence, the new address should not be already used by the middlebox nor should it be assigned to any local device by any other means. The first condition can be checked easily by looking at the context table. The verification of the second condition is more complex, especially if the rewriting use the same prefix for IP_{int} and IP_{ext}. Here, there is a risk of collisions between the randomly generated addresses and addresses already assigned to end devices. One solution could be to check if the address is not already in use, using a neighbour discovery (NDP). But this is unacceptable for at least two reasons. First, it increases the latency for all connection initializations, because the middlebox has to wait until the NDP timed out before making a decision. Second, no response to a NDP request does not mean that this address is not in use, e.g. the device can currently be down. To mitigate the problem, we propose the following:

- The middlebox can be configured to not use the autoconfiguration space derived from the MAC address (this means removing results with 4th byte and 5th byte set to 0xFF and 0xFE respectively). This configuration should be enabled by default to prevent a conflict with the standard configuration;
- In case of DHCPv6 address distribution, the DHCP address space should not be included in the rewriting space configured on the middlebox;

- In case of CGA or static configured addresses, the administrator can manually forbid addresses;
- In any case, the middlebox should maintain a list of devices currently in communication. Clearly this is not exhaustive, since devices can be connected without established connections.

These four rules eliminate the risk of collisions in most networks, and minimize it for some special cases. Additionally, it is important to notice that the risk of collision is actually very low – even without applying the rules mentioned above. The evaluation of the probability of a collision is a variant of the "birthday paradox". Given a pool of n addresses and already j addresses assigned, the probability $\bar{p}(n, j)$ to choose the $j + 1$ address without collision is:

$$\bar{p}(n, j) = 1 - \frac{j}{n} = \frac{n - j}{n} \tag{1}$$

This means that if we assign J addresses in a free space, we have a probability to not have any collision of:

$$\bar{P}(n, J) = \bar{p}(n, 0).\bar{p}(n, 1) \dots \bar{p}(n, J - 1) = \frac{n \cdot (n - 1) \dots (n - J + 1)}{n^J} \tag{2}$$

Then, the probability to have at least one collosion is:

$$P(n, J) = 1 - \bar{P}(n, J) = 1 - \frac{n \cdot (n - 1) \dots (n - J + 1)}{n^J} \tag{3}$$

That we can write:

$$P(n, J) = 1 - \left(1 - \frac{1}{n}\right) \cdot \left(1 - \frac{2}{n}\right) \dots \left(1 - \frac{J - 1}{n}\right) \tag{4}$$

Since for all i in $1 \dots J$: $i \leq J$, we can give an upper bound for the probability with:

$$P(n, J) \leq 1 - \left(1 - \frac{J - 1}{n}\right)^{J-1} \tag{5}$$

We can now perform an evaluation of this probability. In a network with only one prefix of the minimal size for auto-configuration, the interface identifier uses 64 bits. Two bits are reserved for special purpose, so only 62 bits are actually free. We can calculate $n = 2^{62}$. On a big network with one thousand computers, where each of them maintains one thousand flows, we need to allocate $J = 1000 \cdot 1000 = 10^6$ addresses. A simple computation informs us than the probability of collision is less than $2.2 \cdot 10^{-7}$.

4.3 Compatibility Analysis

In order to support wide spread deployment it is essential that our solution integrates smoothly into existing networks. In our case, we only need to deploy the middlebox at the border of the network and an adaptation of the end devices to enable address rewriting. More specific the necessary changes are as follows:

Remote Routers and Servers: since our solution is based on standard IPv6 packets, it is compatible with the standard IPv6 network. There is no need to upgrade intermediate routers or remote servers. We can deploy it locally without cooperation or impact on other networks, the real source address is obfuscated but the packet is still valid.

Local Devices: for local devices, packets without flow label are not rewritten and therefore there are no compatibily implications. But to use the benefits of our solution, upgrades are usually necessary. First, not all Operating Systems (OSes) provide means to set flow labels. Second, on compatible OSes, there applications have to actually use the flow label option. We discuss assignment of flow labels in Section 5.2.

Common Address Translation Issues: Address rewriting is a kind of address translation and can have some consequences. First, some applications send the IP address to the peer within the application layer protocols, for example File Transfer Protocol (FTP) and Session Initiation Protocol (SIP). If transmitting the address at the application layer is mandatory for a given protocol, we can not easily rewrite the address.

For IMCP packets, we have to rewrite the internals addresses in quoted ICMP packet too. It makes parsing a little bit complexer but it does not break ICMP messages.

In case of IPSec, we are in the same case than NPTv6 and the conclusion is the same: peers should detect an address translator, so IPSec should work.

In all cases, our solution is better than standard address translation since it can be easily disabled. For all connections that are not compatible, the applications can set the flow label to zero, the default value.

5 Flow label Assignment at End Devices

5.1 Link between Applications and Operating System: Flow Label Management API

Unfortunately, there is no standard Application Program Interface (API) between a software package and an operating system to set or request a flow label. RFC 3542 [23] is the last standardized API for IPv6, and we can read "This API does not define access to the flow label field, because today there is no standard usage of the field". Without standardized API, each operating system has a specific approach to set and configure flow labels, and it is not easy to write portable software. This is why our work focuses only on the Linux kernel.

The Linux Flow Label API is 13 years old, and part of the Kernel since the version 2.2.7, released in April 1999. Design decisions of the implementation are explained in [2], and there is nothing really new since this time. This document is still the reference documentation for this API.

5.2 How to Patch Applications on Linux

Our proposition is simple: each application discriminates flows and sets flow labels for outgoing sockets, and if it is not done, the operating system has to set a flow label for all sockets of an application. The user can deactivate this kernel behavior (for example with the help of an environment variable).

Thanks to the GNU linker ld, it is possible to preload some libraries for dynamically linked software (the kind of linkage used by nearly all Linux distributions). We can intercept the call to the `connect()` function. Here we check if a flow label is already set, in this case we continue with the standard `connect()` function. Otherwise, we enable the flow label and set its value (derived from the process identifier `PID`) via the `setsockopt()` function.

Using this approach, we do not have to patch every application to use the spreading. At the same time, it does not interfere with a patched application since it does not overwrite an existing flow label value. Even if this is not an optimal solution, the gain is high in comparison with using just one address for all applications running on a given computer.

6 Performance Evaluation

In the following sections, we discuss various performance aspects of our solution.

6.1 Memory Consumption on the Middlebox

Within a context, we have to store:

- the real source address IP_{int} of the computer (128 bits);
- the randomized source address IP_{ext} (128 bits);
- the flow label (20 bits), stored in an integer (32 bits);
- the cached checksum difference C_D (16 bits);
- the "last seen" value, to remove old entries (same size as `jiffies_64` kernel variable, i.e 64 bits);
- two node structures in the hash tables (128 bits each);

The total size is about 80 octets for each context. With the current hypothesis of 1000 computers with 1000 flows each, we need about 80MB to store all contexts. On a home network with 5 computers and 100 flows each, less than 100KB of memory is necessary.

Moroever, each context needs less space than a usual entry of the `conntrack` table used for NAT in IPv4. Therefore, the memory consumption of our solution will not be a problem for modern routers with NAT capacities.

6.2 Latency

Overhead of Rewriting on the Middlebox. Our implementation adds extra computation on the middlebox, before the routing of a packet. This adds some latency for packet treatment. But this computation is simple and it does not show any impact on the latency in our test networks. On a test bed with a standard latency of 300ms, it was not possible to see any impact on the latency.

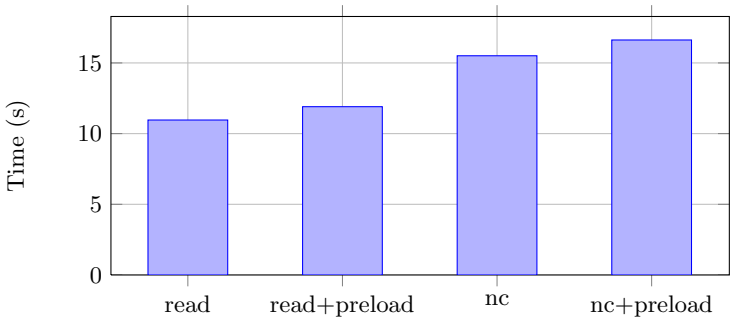

Fig. 2. Average time to run 20.000 times tests with library preload enabled/disabled

Overhead of the Library Preload. A second potential latency cost comes from the library preload, to assign a flow label to each application. Our tests were done on a virtual machine with a 2.4Ghz CPU allocated. We run 20.000 times four different tests:

- open and read a short file. The library preload is not enabled;
- the same command, but with library preload enabled;
- a short `netcat` command, this sends an UDP datagram to localhost and quits, without library preload;
- the same `netcat` command, with library preload enabled.

Figure 2 shows the measured times for the 20.000 iterations. The difference between the two file readings comes from the load of the library at the startup, even if the library does nothing. For each run, it introduces a latency of 5.10^{-5} seconds, but only at the startup of the software (we need to load the library only once). The difference between the two `netcat` tests reveals the total overhead of the library, with the time to set up the flow label included. If we subtract the time to load the library, we get a latency overhead of 9.10^{-6} seconds for each connection, Kernel time included.

As expected, our preload library does not have significant influence on the overall latency.

6.3 CPU Consumption on the Middlebox

To evaluate the CPU consumption of our solution, we made a profiling of the Kernel. We used the OProfile software, part of the standard tools of the Linux Kernel package. Our results show that our module needs about 10% of the time needed by the network card driver itself. With a bandwidth of 2.2Gb/s our module uses less than 2.5% of the CPU time of the 2.4Ghz CPU used.

Additionnaly, we install our solution in a home network on a standard router (Asus-rt16) and we do not see any CPU consumption overhead[2].

[2] The maximal bandwidth of the Internet connection is about 30Mb/s

7 Future Works

Currently our solution allows only to activate or deactivate the address rewriting (encoded by setting the flow label to zero). One of our ideas is to encode more data in the 20 bits of flow labels. The first bit could be used as a flag for a stateful firewall, to allow or deny incoming connections on the ephemeral address. This can be very useful in case of peer-to-peer connections.

Another possibility is to signal to disable the rewriting of the flow label to zero. This can be relevant if an upstream router uses the flow label for Quality of Service. Finally, a configuration of the timeout before erasing a context could be added too. All this flags improve the possibility of the client to adapt to specific communication context. We have enough space to provide five flags, since 32768 flows (15 bits left) should be enough to numbering all flows of a device.

Another improvement can be done in case of multihoming of the middlebox. The flow label tagging allows us to assign one outgoing interface for each flow, thus it allows us to make some load balancing on the middlebox.

8 Conclusion

Our solution provides a privacy gain, since an attacker cannot easily link differents connections of a single computer. It is compatible with all address assignment policies, and is very easy to deploy on a network. The administrator does not need to change the actual configuration and still can use CGA to prevent IP address spoofing on the local network. Our middlebox divides the network in one trusted zone under the control of an administrator and one untrusted zone, where privacy of users is protected.

The intelligence stays on the end devices, able to determine if a flow should be spread or not. On the middlebox, the resource consumption is lower than a typical IPv4 NAT setting and should not be a problem for the deployment. As for a IPv4 NAT, there is a risk of deny of service attacks from local network, by opening a lot of connections with random flow label values. It can be mitigated by limiting the number of allowed connections per devices.

Today, privacy is a hot topic for IPv6. For example, Deutsche Telekom Internet Provider uses privacy as a marketing argument for their IPv6 architecture. A patched middlebox with our solution is relevant in this context.

References

1. World IPv6 launch, `http://www.worldipv6launch.org/` (Consulted the July 4, 2013)
2. Kuznetsov Alexey, N.: IPv6 flow labels in Linux-2.2, Tech. report, Institute for Nuclear Research, Moscow (April 1999)
3. Alsa'deh, A., Rafiee, H., Meinel, C.: Stopping time condition for practical. In: 2012 International Conference on IPv6 Cryptographically Generated Addresses,Information Networking (ICOIN), pp. 257–262 (February 2012)

4. Amante, S., Carpenter, B., Jiang, S.: Rationale for Update to the IPv6 Flow Label Specification, RFC 6436 (Informational) (November 2011)
5. Amante, S., Carpenter, B., Jiang, S., Rajahalme, J.: IPv6 Flow Label Specification, RFC 6437 (Proposed Standard) (November 2011)
6. Arkko, J., Kempf, J., Zill, B., Nikander, P.: SEcure Neighbor Discovery (SEND), RFC 3971 (Proposed Standard) (March 2005), Updated by RFCs 6494, 6495
7. Audet, F., Jennings, C.: Network Address Translation (NAT) Behavioral Requirements for Unicast UDP, RFC 4787 (Best Current Practice) (January 2007)
8. Aura, T.: Cryptographically Generated Addresses (CGA), RFC 3972 (Proposed Standard) (March 2005), Updated by RFCs 4581, 4982
9. Berthold, O., Federrath, H., Köpsell, S.: Web mixes: A system for anonymous and unobservable internet access. In: Federrath, H. (ed.) Anonymity 2000. LNCS, vol. 2009, pp. 115–129. Springer, Heidelberg (2001)
10. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
11. Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM 24(2), 84–90 (1981)
12. Van de Velde, G., Popoviciu, C., Chown, T., Bonness, O., Hahn, C.: IPv6 Unicast Address Assignment Considerations, RFC 5375 (Informational) (December 2008)
13. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Proceedings of the 13 th Usenix Security Symposium (2004)
14. Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C., Carney, M.: Dynamic Host Configuration Protocol for IPv6 (DHCPv6), RFC 3315 (Proposed Standard) (July 2003), Updated by RFCs 4361, 5494, 6221, 6422
15. Guha, S., Biswas, K., Ford, B., Sivakumar, S., Srisuresh, P.: NAT Behavioral Requirements for TCP, RFC 5382 (Best Current Practice) (October 2008)
16. Hain, T.: Architectural Implications of NAT, RFC 2993 (Informational) (November 2000)
17. Hinden, R., Deering, S.: IP Version 6 Addressing Architecture, RFC 4291 (Draft Standard) (February 2006), Updated by RFCs 5952, 6052
18. Hu, Q., Carpenter, B.: Survey of Proposed Use Cases for the IPv6 Flow Label, RFC 6294 (Informational) (June 2011)
19. Lindqvist, J.: IPv6 is bad for your privacy, Defcon 15 (2007)
20. Narten, T., Draves, R., Krishnan, S.: Privacy Extensions for Stateless Address Autoconfiguration in IPv6, RFC 4941 (Draft Standard) (September 2007)
21. Rijsinghani, A.: Computation of the Internet Checksum via Incremental Update, RFC 1624 (Informational) (May 1994)
22. Srisuresh, P., Holdrege, M.: IP Network Address Translator (NAT) Terminology and Considerations, RFC 2663 (Informational) (August 1999)
23. Stevens, W., Thomas, M., Nordmark, E., Jinmei, T.: Advanced Sockets Application Program Interface (API) for IPv6, RFC 3542 (Informational) (May 2003)
24. Wasserman, M., Baker, F.: IPv6-to-IPv6 Network Prefix Translation, RFC 6296 (Experimental) (June 2011)