Luth: Composing and Parallelizing Midpoint Inspection Devices

Ion Alberdi, Vincent Nicomette, Philippe Owezarski
CNRS ; LAAS ; 7 avenue du Colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS , INSA , INP, ISAE ; LAAS ; F-31077 Toulouse, France
Email: ialberdi@laas.fr

***Abstract*—The race for innovation is driving Internet evolution. Internet software developers have to create more complex systems while enduring the pressuring time to market. Therefore, end-host software have bugs, vulnerabilities and cannot be trusted. That's why, among others, network Intrusion Detection System (IDS), Intrusion Prevention System (IPS), firewall or other network devices monitor such software to prevent unexpected behaviors. However, their functionalities are limited by design, because they can only handle a configuration of predefined monolithic protocol layerings. In this paper we present Luth, a midpoint inspection device that relies on the composition and parallelization of predefined midpoint inspectors (MI). We present the main functionalities offered by its configuration language and interpreter. Finally, we benchmark a prototype implemented in OCaml. This prototype runs in the userspace of a GNU/Linux operating system, by means of the libnetfilter_queue library. We show how it efficiently inspects and filters DNS hidden-channels encapsulated into 20 GRE tunnels.**

***Keywords*-Internet, IPS, IDS, Firewall.**

I. Introduction

The race for innovation is driving Internet evolution. Internet software developers then have to create more complex systems while enduring the pressuring time to market. Therefore, end-host software have bugs, vulnerabilities and cannot be trusted. That's why, among others, diverse network devices, we generalize into the midpoint inspection devices term, have been designed. Midpoint inspection devices monitor end-host software to prevent unexpected behaviors resulting from Internet malicious activities. For example the packet filters i.e, a "programmable selection criterion for classifying or selecting packets from a packet stream in a generic, reusable fashion" [1] mainly focus their work on the performance of the packet selection [1], [2], [3]. Intrusion Detection Systems (IDS) [4], [5] try to detect network attacks, when Intrusion Prevention Systems (IPS) [6] try to prevent the consequences of attacks [7], [8]. Stateless and stateful firewalls aim at implementing a network policy, that states, among others, which network services can be accessed by which client [9], [10]. Such firewalls are driven by languages and associated interpreters that can lead to buggy configurations. Therefore, studies in [11], [12], [13], address the problem of the coherence of existing stateless firewall rules. The configuration of stateful firewalls is addressed in [14]. Application detectors aim at detecting the applicative protocols involved in a communication [15]. Finally, some languages are proposed to help creating new IDS, IPS, firewall, application detectors, or other midpoint inspection devices [16], [17].

We have a main criticism to formulate against all these solutions. All these devices are based on the configuration of a set of predefined layerings. In other words, this approach proposes to configure *monolithic protocol layerings*. The research done in network protocol architectures has lead to what are called, dynamically configurable protocol layerings [18], [19], [20], [21]. Such solutions split one protocol into smaller primitive building blocks, which can then be composed by a configuration language. The idea described by the language in the FFPF packet filter [2], allows different filters to be composed too. This is up to our knowledge the only solution to configure a non predefined set of protocol layerings. However, their solution does not provide algorithms to check neither the correctness of the configured layering, nor the factorization of unnecessarily duplicated computation. That's why we designed Luth, the tool we present in this article. In section 2 we describe more in detail the limits of the monolithic approach, to motivate and introduce the need for a different approach. We present then our contribution. Luth proposes a configuration language and its interpreter to correctly and efficiently compose and parallelize a given set of protocol inspectors. In other words, providing a correct policy is written in its configuration language, Luth computes and executes an inspection algorithm. In section 4 we describe an online experiment in which Luth filters DNS hidden channels encapsulated into 20 GRE tunnels. The conclusion of the article summarizes our contribution and describes future works.

II. The Limits of the Monolithic Approach

Before going further we need to introduce the '$/$' operator that composes interoperable midpoint protocol inspectors. For example, we illustrate a **stateless** firewall rule accepting network packets having the source 192.168.0.1 IPv4 address and carrying TCP segment with destination port 22 as: IPv4(src=192.168.0.1)$/$TCP(dport=22).

Let us illustrate more in detail the differences between the monolithic approach, implemented by a tool $T1$, and the approach we propose, implemented by a tool $T2$.

$T1$ provides inspection capabilities that can be configured by a set of parameters pIJ, related to some layerings like $SL = \{$
$S11(p11)/.../S1J_1(p1J_1), ..,$
$SI1(pI1)/.../SIJ_I(pI_I), ..,$
$SN1(pN1)/.../SNJ_N(pNJ_N)\}$.

Nowadays the network layerings and communications are more and more complex and diversified, for at least two reasons:

1) The encapsulation of protocols are widely used to implement virtual private networks, or to evade inspection devices. For example actual IDS, IPS or firewalls can be bypassed by implementing new protocol stacks over HTTP[1] or DNS[2] among others.
2) The variety of software that implements different services in the Internet. Each software has its own bugs and should require a special and adapted inspection algorithm. For example, even if there is a single TCP protocol, there is at least a particular TCP implementation by operating system. Therefore, each implementation presents different properties and bugs. For example, it is useless to prevent the exploitation of the MS09-048 vulnerability present in the TCP stack of some versions of Microsoft Windows operating systems[3], when inspecting TCP segments towards a server running on an OpenBSD system. Similarly, even if there is a single IPv6 protocol, it is useless to inspect the possible exploitations of the CVE-2007-1365 vulnerability, targeting the IPv6 stack of a version of the OpenBSD operating system, when inspecting IPv6 packets towards a Microsoft Windows operating system. Moreover, it is useless to drop the exploitations of a vulnerability targeting one protocol stack implementation if the application we are monitoring uses a patched and corrected version of the later.

Thus, the inspection of the sequence of network messages sent by different applications that use different protocol layerings, requires to implement very different and flexible policies. We do think the monolithic layering approach is not suited to this situation for two reasons:

1) The encapsulation of protocols creates circular dependencies. Indeed, the monolithic approaches implement the composition of protocols by hard-coded function calls, linked imperatively [10] or using the inheritance offered by object languages [5]. Let us try to implement the layering IPv4/HTTP/HTTPEncaps/IPv4/..., that encapsulates IPv4 packets inside HTTP messages to bypass a given midpoint inspection device. To implement such layering, the HTTPEncaps inspection function needs to call the one related to IPv4. Therefore, HTTPEncaps depends on IPv4. Applying the reasoning to the whole layering, we have IPv4 that depends on HTTP (IPv4/HTTP). Then, by transitivity HTTP depends on IPv4 (HTTP/HTTPEncaps/IPv4). Finally, by transitivity again, each protocol inspector function depends on each other. Such kind of circular dependencies are difficult to implement, conducting to messy and hardly extensible implementations. Among others, we do think it is the reason why the solutions to handle the GRE encapsulations are limited nowadays. Indeed such solutions either:
   - do not inspect protocols encapsulated inside a tunnel [10],
   - handle a single encapsulation [4],
   - do not advertise the possibilities related to the inspection of encapsulated layerings [7], [22], [5].
2) It requires more work to implement both the relations between protocol inspectors **and** protocol inspectors themselves, than implementing protocol inspectors only. We do think that is why the variety of available protocol inspectors is limited and not adapted to the diversity of the situations to inspect.

Instead of relying on the configuration of a sequence of monolithic protocol layerings, the approach we propose is based on a set of midpoint inspectors, MI, that can be:

- configured by the parameters related to each MI, $MISET = \{MI1(p1), .., MIi(pi), .., MIN(pN)\}$,
- composed by means of the $/$ operator,
- parallelized by means of the $|$ operator. For example, we illustrate a **stateless** firewall rule accepting network packets having the source 192.168.0.1 IPv4 address, carrying:
  - TCP segments with destination port 22, or
  - UDP datagrams with destination port 53, as

  IPv4(src=192.168.0.1)$/$[TCP(dport=22)$|$UDP(dport=53)].

Let us take a more concrete case, involving the $IPv4$, GRE, TCP and $HTTP$ protocols. Let us say the midpoint inspector needs to perform a policy that states:

1) The two extremities 192.168.0.1 and 192.168.0.2 encapsulate packets into two GRE tunnels. In the encapsulated networks, only TCP segments should be present in the packets from 10.0.1.0/24, and 10.0.2.0/24. Moreover, only $HTTP$ requests should be contained in the segments from 10.0.2.0/24, and an IDS should launch an alarm when a host is trying to send overlapping TCP segments to evade the application level inspections [23].
2) One GRE tunnel is encapsulated from 192.168.1.1 and 192.168.1.2, where encapsulated packets belong to the network 10.0.0.0/8. The clients accessing the HTTP server in 10.0.0.1 should use normalized TCP sessions, and a specific IDS algorithm should monitor

[1]http://www.http-tunnel.com/html
[2]http://www.hsc.fr/ressources/outils/dns2tcp
[3]http://www.microsoft.com/technet/security/bulletin/ms09-048.mspx

the applicative communications related to the web service hosted in 10.0.0.1. The clients towards the server 10.0.0.2 should speak by means of the TCP protocol towards the server port 22.

To do so, the tool $T1$ should develop 4 different layerings and, considering a "default drop" policy, write the following sequence of rules:

```
/* first policy rule */
IPv4(
extrm1=192.168.0.1,
extrm2=192.168.0.2)/GRE/IPv4/GRE/
IPv4(net=10.0.2.0/24)/TCP(mode=only\_tcp);
IPv4(
extrm1=192.168.0.1,
extrm2=192.168.0.2)/GRE/IPv4/GRE/
IPv4(net=10.0.1.0/24)/
TCP(mode=reassembly_ids)/
HTTP(mode=rfc_compliance);
/* second policy rule */
IPv4(
extrm1=192.168.1.1,
extrm2=192.168.1.2)/GRE/
IPv4(net=10.0.0/8)/
TCP(mode=stream_normalization,
srv_addr=10.0.0.1)/
HTTP(mode=specific_ids);
IPv4(
extrm1=192.168.1.1,
extrm2=192.168.1.2)/GRE/
IPv4(net=10.0.0/8)/
TCP(srv_addr=10.0.0.2,srv_port=22)
```

To do so, the tool $T1$ should implement the protocol inspectors:

- $IPv4$ with the parameters $extrm1, extrm1, net$,
- GRE,
- TCP with the parameters $mode, srv\_addr, srv\_port$,
- $HTTP$ with the parameters $mode$.

Moreover, the tool should implement the links involved in the different layerings, which are in this case 4, i.e:

- $IPv4/GRE/IPv4/GRE/IPv4/TCP$,
- $IPv4/GRE/IPv4/GRE/IPv4/TCP/HTTP$,
- $IPv4/GRE/IPv4/TCP$,
- $IPv4/GRE/IPv4/TCP/HTTP$,

By using the approach based on the composition and parallelization of parameterized MIs, the tool $T2$ could be configured like this:

Listing 1: Configuration of T2

```
[
IPv4(
extrm1=192.168.0.1,
extrm2=192.168.0.2)/GRE/IPv4/GRE/
[IPv4(net=10.0.2.0/24)/TCP(mode=only_tcp) |
IPv4(net=10.0.1.0/24)/
TCP(mode=reassembly_ids)/
HTTP(mode=rfc_compliance)
] |
IPv4(
extrm1=192.168.1.1,
extrm2=192.168.1.2)/GRE/
IPv4(net=10.0.0/8)/ |
[TCP(mode=stream_normalization,
srv_addr=10.0.0.1)/
HTTP(mode=specific_ids) |
TCP(srv_addr=10.0.0.2,srv_port=22) ]
]
```

To do so, $T2$ should develop the same number of protocol inspectors as in $T1$, and implement the generic composition and parallelization of protocol inspectors. The need for implementing such generic relations can be motivated by a similar policy where the number of GRE tunnels is different. For example, if the number of tunnels is reduced to zero, the tool $T1$ should implement two different layerings, i.e

- $IPv4/TCP$,
- $IPv4/TCP/HTTP$,

when the tool $T2$ is already capable of implementing it.

III. Main Ideas Behind the Interpreter of the Configuration Language

In this section, for the sake of brevity, we only present the main ideas behind the interpreter of the configuration language. For a more detailed description, the reader is invited to read [24]. The interpreter of the configuration language tries to create a valid and optimized tree from the configuration rule. Then, an inspection algorithm is deduced by browsing and updating this tree, implementing the state of the inspection. The basic element of our systems are Midpoint Inspectors (MI). The interpreter offers three different features to analyze the deduced inspection tree:

- the validity of inspection trees,
- the missing default layers of an incomplete layering,
- the optimized version of valid inspection trees, that factorizes duplicated father.

The input and output interfaces of an MI are defined by their types. Then, analyzing this information, the interpreter verifies whether the rule is correct and deduces an inspection tree. For example the rule in Listing 1 is depicted in Figure 1.

Let us illustrate how the inspection algorithm behaves by illlustrating the inspection of the Gre node emphasized on Figure 1. Let us say its father, i.e an $IPv4$ node, outputs a TCP segment contained in its payload. The Gre node is expecting a GRE packet; therefore it sends back the $drop$ decision to its father without consulting its children. Let us say its father sends him a GRE packet to analyze. The Gre node extracts its payload and sends the new message to its two $IPv4$ children. The latter recursively inspect the packet. Each of the children send back a decision, $ds1$ and $ds2$, to their father, i.e the Gre node. To compute a decision from a set of decisions, the inspection algorithms uses a $reduce$ function. The only condition such a function needs to fulfill so that factorized

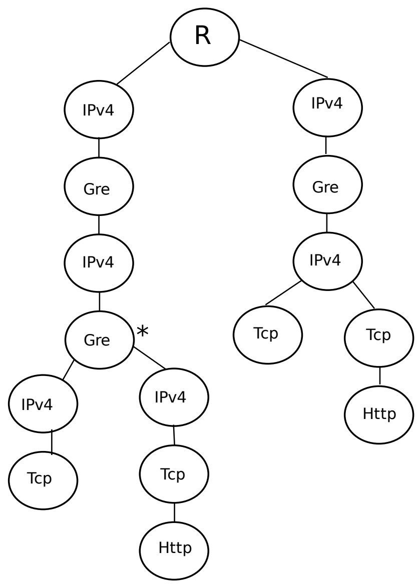

Figure 1: Inspection tree deduced from the rule in Listing 1.

and non factorized trees perform similar computations is the following: $\forall X \in \wp(\wp(\mathbf{DS})), reduce(\{reduce(x), x \in X\}) = reduce(\bigcup_{x \in X} x)$ [24]. We use an analogy with election algorithms to illustrate this property. In indirect elections, people first vote for special electors, who, in a second round, make the final choice. By taking as X, the set of people electing their special electors, the election of super electors can be seen as $\{reduce(x), x \in X\}$, and the choice of super electors as $reduce(\{reduce(x), x \in X\})$. In direct election systems, all the people vote at once which can be seen as $reduce(\bigcup_{x \in X} x)$. The property of this reduce function aims at imposing that the same decision is taken using direct or indirect algorithms. Coming back to our inspection tree, the Gre node finally sends back the decision $reduce(\{ds1, ds2\})$ to its father.

IV. Implementation and Benchmarking of the Midpoint Inspection Device

This section presents Luth, the midpoint inspection device prototype we developed.

A. Name and Targeted Platform

Luth is a new kind of midpoint inspection device that can be configured with great flexibility. Inspired by the analogy made by computer firewalls, Luth stands for Lur Ur Ta Haize, which means in Basque language, Earth, Water And Wind. Indeed, fire can be stopped by water and earth or propagated by wind. The same way Luth can use different ways to inspect Internet communications, or to let some anomalies bypass itself. The implementation of Luth is written in $OCaml$, making a quite extensive use of the Melange library to safely and efficiently parse packets [25].

To perform online inspections, the binding of the libnet-fliter_queue [10] to OCaml has been developed with the help of the stub code generator CamlIDL[4]. This library offers some system-calls to communicate with Netfilter, GNU/Linux's framework to implement firewalls and NAT routers. To check the correction and measure the performance of the current implementations we perform an experiment addressing DNS tunneling.

B. Inspecting DNS hidden-channels

To get the Internet for free in commercial hotspots, a hidden-channel can be established using the DNS protocol. Indeed, to redirect new clients' first web page requests to a portal where the payment system is explained, this kind of hotspot needs to let the client make DNS[5] requests. However, whereas this service only needs to make hostname resolutions, the hotspot usually let the users ask all types of DNS requests. Users trying to get Internet for free use the semantically vast enough TXT[6] requests, to encode TCP segment into DNS messages [26]. To sum up, those bypassing layerings can be seen as IPv4/UDP/DNS(messages=TXT)/Tunnel/... Therefore, to remove such channels one just needs to drop TXT requests.

1) Experimental Setup: To stress our firewall in this situation we choose to install the $dnsmasq$[7] DNS forwarder in the server. $dnsmasq$ forwards all the DNS requests it receives to a real DNS server. We then bind the $apache2$[8] web server on the address 192.168.10.11, and generate 20 HTML pages. Each of these pages have 10 images. Those images size follows a 1.2 pareto distribution [27]. We multiply the pareto coefficients by 10240 to obtain image sizes from 10 to 500KB. The simulation software in the legitimate client makes a host name resolution request to a predefined set of domain names, and then downloads one of the 10 web pages only if the request is successful.

To implement the hotspot thief, we register a domain name to make the DNS tunneling possible. We choose the *dns2tcp* [26] tunneling framework and install the client software in the client with the address 192.168.2.10 (Figure 2). Then, we simulate a client that periodically uses the SSH protocol over this DNS tunnel. After getting a shell on the distant server, we ask the server to display the contents of one of the 100 files generated with the same pareto distribution using the cat utility. With a coefficient of 10240, we obtain file sizes from 10 to 189 KB. We associate a timeout of 10s for the completion of these requests. To demonstrate that

[4]http://caml.inria.fr/pub/old_caml_site/camlidl
[5]http://www.ietf.org/rfc/rfc1034.txt
[6]http://www.isi.edu/in-notes/rfc1464.txt
[7]http://thekelleys.org.uk/dnsmasq/doc.html
[8]http://httpd.apache.org/

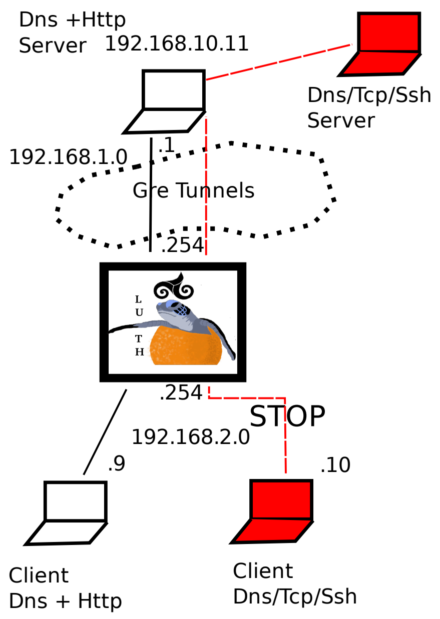

Figure 2: Tunneled DNS filtering. The 192.168.2.9 client downloads files from the HTTP server in 192.168.10.11 by means of DNS and HTTP protocols. The malicious client in 192.168.2.10 uses a hidden channel by means of the DNS protocol to download files from the SSH server.

Network interface cards (NIC)	Gigabit Ethernet controllers: Broadcom 5721J and IntelPro1000PT
Interconnection	Gigabit Ethernet: Cisco Catalyst 6504
Processors	Dual Core Xeon 3050
Memory	2GB of 667MHz Dual Rank ECC Memory (2x1GB)
Operating System	Debian GNU/Linux (2.6.18 kernel)

Table I: Computer and interconnection network's characteristics.

Luth can handle cyclic layerings, we implement the filtering of this scenario into self-encapsulated GRE [28] tunnels as depicted in Figure 2[9]. All the different hosts are present in *laasnetexp*, a testbed environment in our laboratory [29]. The computers used in this experiment present the characteristics described on Table I.

The server and the firewall, i.e the server executing Luth, are connected by the 192.168.1.0/24 network. To reach the 192.168.10.0/24 network, the firewall needs to go through n *Generic Routing Encapsulation* (GRE) [28] tunnels. The server, has to go through the same tunnels to reach the 192.168.2.0/24 network. Finally the clients use the 192.168.2.254 gateway to reach the 192.168.10.0/24 network. The implementation of the n encapsulations is achieved using the *iproute* package of the Debian project[10]. We configure $n = 20$ tunnels between the firewall and the server.

To stress our firewall, we first randomly generate a scenario: 5 clients download on each host, random files during one minute every t, where t follows a law having an exponential distribution of mean 0.01ms. We measure for each download, the time from the start of the client application to the end of it, divided by the size of the downloaded file. This measure is noted download bandwidth. For a failed download the client returns -1 as download bandwidth.

We run the experiments in three cases studies.

- First, no firewall is put in the midpoint inspection device, labeled Ref.
- Second, we put a userspace firewall we have written for benchmarking purposes in C, labeled Cfw. This firewall accepts all the packets after having read them by means of of the libnetfilter_queue library.
- Third, we put Luth configured with a certain rule in it, labeled $Luth$.

2) Results Using a Default Configuration: We configure Luth using the rule depicted in Listing 2. For the sake of brevity, instead of $Tcp(mode = stream\_normalization)$, $Dns(mode = no\_hidden\_channel)$, and $Http(mode = rfc\_compliance)$ we write the MIs, Tcp, Dns and $Http$.

Listing 2: DNS rule

```
IF(name=idx2,addrs=default,messy=yes,
   lo=192.168.1.254);;
INSP(L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
     Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
     Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
     Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
     Gre/Ipv4/Gre/Ipv4/Gre/[ Dns | Http ]);;
```

The filtering tree computed by the firewall which fills the holes between Dns $Http$ and Gre MIs is depicted in Listing 3.

Listing 3: DNS rule's MI Tree

```
L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
Gre/Ipv4/Ttl(min_ttl=10)/
 [Tcp/[TPort(sr_port=53)/Dns |
       TPort(sr_port=80)/Http] |
  Udp/UPort(sr_port=53)/Dns]
```

Figure 3 shows the bandwidth of the first of the five unauthorized clients without Luth (other clients present similar results). With Luth all malicious downloads output -1,

[9]The reader should notice that these tunnels are not necessary for both clients to perform their downloads. Indeed, the tunnels only serve to illustrate the expressiveness of Luth.

[10]http://packages.debian.org/search?keywords=iproute

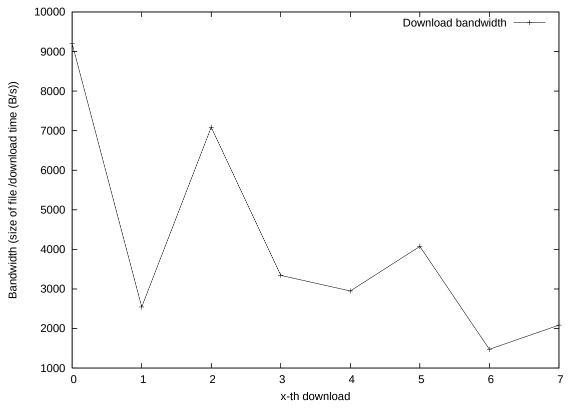

Figure 3: Performance of malicious DNS clients in the unfiltered case.

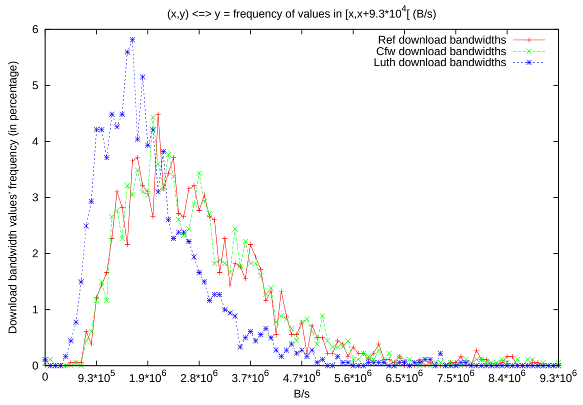

Figure 4: Distribution of downloads bandwidth obtained with DNS filtering using default configuration. The download bandwidths reached in Ref and Cfw experiments are mostly in the interval $[1.9*10^6, 2.8*10^6[$ B/s, when the ones obtained during Luth experiment are in $[9.3*10^5, 1.9*10^6[$.

showing all such requests have successfully been dropped. The five legitimate clients perform about 2000 downloads during the experiment. Their download bandwidth reaches $10^6 B/s$, going from 1 up to $10^7 B/s$ for the fastest downloads during the experiment without midpoint inspectors. Figure 4 shows the frequency of the downloads in the different slots from 0 to 9.3 MB/s. We see that the curve labeled Luth seems to be the translation to the left of both other curves, meaning that the bandwidths obtained with Luth, are overall slower than in both the other cases.

By comparing Luth's bandwidth curve, to Cfw and Ref curve, both the side-effects due to being in userspace and due to the performed inspections are taken into account. However, we want to just evaluate the side-effects due to the performed inspections. To have a better understanding of the difference among these three distributions, Figure 5 plots the following distributions representing the 2000 legitimate client downloads times:

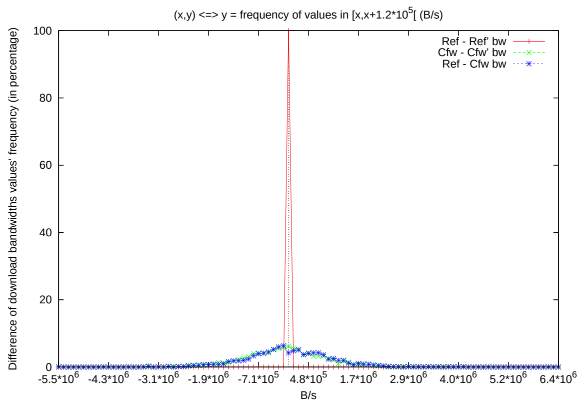

Figure 5: Evaluation of the side effect provoked by a userspace firewall. Being in userspace creates a variability that influences the bandwidth results as much as the performed inspection $((Cfw - Cfw' \simeq Ref - Cfw) \neq Ref - Ref')$.

- Ref - Ref': Ref' being the experiment Ref executed another time.
- Cfw - Cfw': Cfw' being the experiment Cfw executed another time.
- Ref - Cfw.

We see in Figure 5 that the difference of Ref - Ref' is negligible: the network state, and respective client and server application states are stable enough during the two experiments. The negative results obtained in Ref - Cfw shows that some downloads have been faster in the experiment using the firewall written in C, than in the Ref experiment. One explanation of such a trend is that when the userspace firewall written in C slows down some downloads, the following downloads reach the HTTP server faster and are therefore served more quickly than in the Ref experiment. Even if interesting, we have not investigated this phenomenon more deeply. What should be noticed here is that Cfw - Cfw' looks like Ref - Cfw. By this, we see that the scheduling, queueing, and context-switching, induced by the userspace firewall creates a non negligible variability, independent of the inspections done by the firewall (else Cfw - Cfw' will look like Ref - Ref').

To isolate userspace side-effects and measure the ones due to Luth inspections, we compare the difference between Luth and Cfw and between Luth and Ref, with the differences between Cfw and Cfw', or Ref and Cfw. Indeed, the latter two curves, i.e Cfw - Cfw' and Ref - Cfw, exhibit the side-effects due to being in userspace. That's why Figure 6 illustrates the following plots:

- Cfw - Cfw',
- Ref - Cfw,
- Ref - Luth,

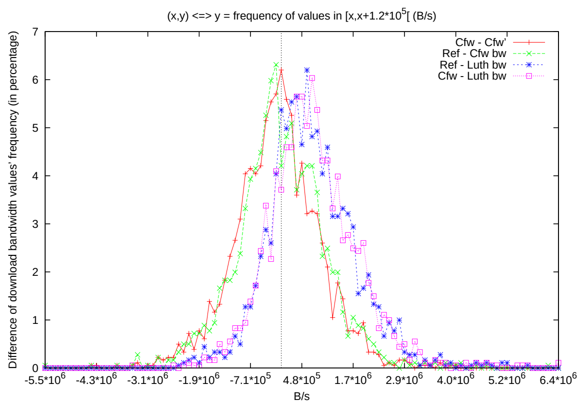

Figure 6: Distribution of download bandwidth differences in DNS experiment with default configuration.

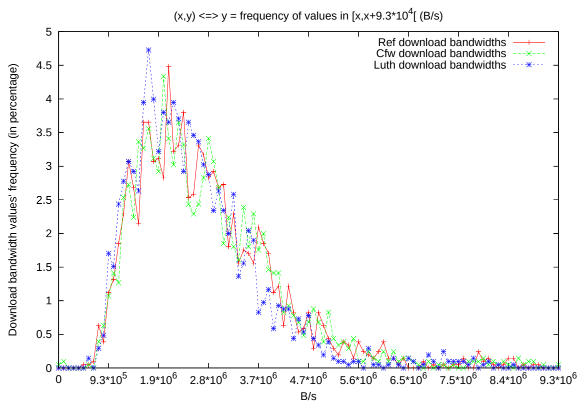

Figure 7: Distribution of downloads bandwidth obtained with DNS filtering using optimized configuration.

- Cfw - Luth.

As depicted in Figure 6, the curves Cfw - Luth, and Ref - Luth have similar shapes. The same way the curves Cfw - Cfw' and Ref - Cfw present similar shapes, that seem to be a translation from about 1MB/s to the left. This translation shows a non negligible slow down induced by the inspections performed by Luth in its default and non adapted configuration.

3) Results Using a Custom Configuration: The rule in Listing 4 aims at specifying a more adapted policy. If the computations done by the hotspot's HTTP server are trusted[11], inspecting the layering involved in the HTTP session (TCP/HTTP) can be considered as useless. In this case, we can configure Luth to accept all TCP segments, without performing any inspection on them, and concentrate the work on DNS filtering. This policy can be written using the rule depicted in Listing 4.

Listing 4: Dns custom rule

```
IF(name=idx2,addrs=default,messy=yes,
   lo=192.168.1.254);;
INSP(L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/
Ipv4/Gre/Ipv4/
[Tcp(mode=tcp_only)|Udp/Dns]);;
```

The figures 7 and 8 show that with this configuration, the slow down induced by Luth's inspection is less visible.

In this case study, we have shown that Luth could be used to efficiently implement a policy that drops DNS tunneling in open hot spots.

V. Conclusion

We have presented in this paper the tool we use to monitor malicious communications. The expressiveness of this midpoint inspection device enables to monitor, up to our knowledge, all Internet protocol layerings while providing correction and optimality checking. The DNS case study shows that Luth, the userspace prototype of our midpoint inspection device, correctly implements policies involving 20 iterations of a cyclic layering. The most promising future work is the parallelization of the inspection algorithm. Indeed, our inspection algorithm is similar to MapReduce algorithms used by Google [30] and could therefore be easily parallelized, i.e instead of waiting the answer of the first child before asking the second child, an improvement consists in asking all the children to inspect the output messages in parallel.

[11]Indeed in this peculiar case, the DNS tunneling problem comes from the DNS server.

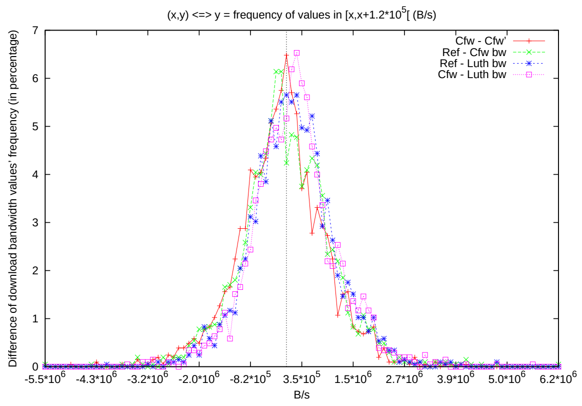

Figure 8: Distribution of download bandwidth differences in DNS experiment with optimized configuration.

References

[1] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM, 1999, pp. 123–134.

[2] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "Ffpf: fairly fast packet filters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 24–24.

[3] Z. Wu, M. Xie, and H. Wang, "Swift: a fast dynamic packet filter," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 279–292.

[4] B. Caswell, J. C. Foster, R. Russell, J. Beale, and J. Posluns, *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[5] V. Paxson, "Bro: a system for detecting network intruders in real-time," in *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 1998, pp. 3–3.

[6] X. Zhang, C. Li, and W. Zheng, "Intrusion prevention system design," in *CIT '04: Proceedings of the The Fourth International Conference on Computer and Information Technology*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 386–390.

[7] Juniper, "Safeguarding your network ips overview demo," http://www.juniper.net/products_and_services/intrusion_prevention_s%olutions.

[8] McAfee, "Host intrustion prevention for server," http://www.mcafee.com/us/enterprise/products/host\intrusion\prevention/host\intrusion\prevention\server.

[9] "Pf: The openbsd packet filter," http://www.openbsd.org/faq/pf/.

[10] "Netfilter, firewalling, nat and packet mangling for linux," http://www.netfilter.org/.

[11] M. G. Gouda and A. X. Liu, "Structured firewall design," *Comput. Netw.*, vol. 51, no. 4, pp. 1106–1120, 2007.

[12] A. Liu, "Formal verification of firewall policies," in *Communications, 2008. ICC '08. IEEE International Conference on*, May 2008, pp. 1494–1498.

[13] A. X. Liu and M. G. Gouda, "Diverse firewall design," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 19, no. 9, pp. 1237–1251, Sept. 2008.

[14] M. Gouda and A. Liu, "A model of stateful firewalls and its properties," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, June-1 July 2005, pp. 128–137.

[15] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic application-layer protocol analysis for network intrusion detection," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[16] N. Borisov, D. Brumley, H. Wang, J. Dunagan, P. Joshi, and C. Guo, "A generic application-level protocol analyzer and its language," in *14th Annual Network & Distributed System Security Symposium*, 2007. [Online]. Available: http://research.microsoft.com/pubs/70223/tr-2005-133.pdf

[17] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," *ACM Trans. Comput. Syst.*, vol. 22, no. 4, pp. 381–420, 2004.

[18] N. T. Bhatti and R. D. Schlichting, "Configurable communication protocols for mobile computing," in *ISADS '99: Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 220.

[19] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: a system for constructing fine-grain configurable communication services," *ACM Trans. Comput. Syst.*, vol. 16, no. 4, pp. 321–366, 1998.

[20] N. Hutchinson and L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

[21] E. Exposito, P. Senac, and M. Diaz, "Fptp: the xqos aware and fully programmable transport protocol," in *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, Sept.-1 Oct. 2003, pp. 249–254.

[22] Cisco, "Ips solutions," http://www.cisco.com/en/US/netsol/ns785/networking_solutions_package.ht%ml.

[23] N. J., S. S., and S. fire Incorporated, "Target based tcp stream reassembly," 2007.

[24] I. Alberdi, "Malicious trafic observation using a framework to compose and parallelize midpoint inspection devices," Ph.D. dissertation, University of Toulouse, INSA-TOULOUSE, 2010.

[25] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan, "Melange: creating a "functional" internet," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 101–114.

[26] HSC, http://www.hsc.fr/ressources/outils/dns2tcp/.

[27] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 835–846, 1997.

[28] "Rfc2784: Generic routing encapsulation (gre)," http://www.faqs.org/rfcs/rfc2784.html.

[29] P. Owezarski, P. Berthou, Y. Labit, and D. Gauchard, "Laasnetexp: a generic polymorphic platform for network emulation and experiments," in *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 1–9.

[30] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.