POLITECNICO DI TORINO
Repository ISTITUZIONALE

Enforcement of dynamic HTTP policies on resource-constrained residential gateways

Original
Enforcement of dynamic HTTP policies on resource-constrained residential gateways / Bonafiglia, R., Sapio, A., Baldi, M., Risso, F.G.O., Pomi, P.C.. - In: COMPUTER NETWORKS. - ISSN 1389-1286. - STAMPA. - 123:(2017), pp. 169-183. [10.1016/j.comnet.2017.05.016]

Availability:
This version is available at: 11583/2679585 since: 2017-09-10T01:22:07Z

Publisher:
Elsevier

Published
DOI:10.1016/j.comnet.2017.05.016

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

13 October 2026

Enforcement of Dynamic HTTP Policies on Resource-constrained Residential Gateways

Roberto Bonafiglia[a], Amedeo Sapio[a,*], Mario Baldi[a], Fulvio Risso[a], Paolo C. Pomi[b]

[a] *Department of Control and Computer Engineering, Politecnico di Torino, Italy*
[b] *Ennova, Torino, Italy*

Abstract

Given that nowadays users access content mostly through mobile apps and web services, both based on HTTP, several filtering applications, such as parental control, malware detection, and corporate policy enforcement, require inspecting Universal Resource Locators (URLs) contained in HTTP requests. Currently, such filtering is most commonly performed in end devices or in middleboxes. Filtering applications running on end devices are less resource intensive because they operate only on traffic from a single user and possibly leverage a hook at the HTTP level to access protocol data, but it is left to the user whether to execute them. On the other hand, middleboxes present the challenge of ensuring that they lay on the path of all the traffic from any relevant device. Residential gateways seem to be the ideal place where to implement traffic filtering because they forward all traffic generated by the hosts on home(-office) networks. However, these devices usually have very limited computation and memory resources, while URL-based filtering is quite demanding. In fact existing approaches rely on a large database of rules coupled with either deep packet inspection or transparent proxying for URL extraction.

This paper introduces *U-Filter*, a URL filtering solution based on a distributed architecture where a lightweight, efficient URL extraction and policy enforcement component runs on residential gateways, delegating to a remote

*Corresponding author. Email address: amedeo.sapio@polito.it.

policy server the resource intensive task of verifying policy compliance. Thanks to the lightweight communication between the two components and the very limited resource requirements of the local module, U-Filter (*i*) can be deployed on resource-limited devices such as residential gateways, and (*ii*) has almost no impact on the performance of the device, as well as on the users' browsing experience, as demonstrated by the experiments presented in the paper.

Keywords: Deep packet inspection, Policy enforcement, Residential gateway, URL filtering

---

1. Introduction

Modern residential gateways are widely deployed to provide broadband Internet access to families, small and medium-sized enterprises supporting a wide range of data rates, from a few Mbps up to 1 Gbps [1]. The architecture of residential gateways is characterized by special purpose hardware chips that forward packets at high speed at the data link layer, while general-purpose components, such as CPU and central memory, are usually employed for other operations that require more sophisticated processing. Since all the traffic directed to Internet hosts (i.e., outside the residential or corporate branch network) must pass through the residential gateway, it is the ideal appliance to apply traffic filtering. Hence, its processing capabilities, often underutilized, could be leveraged by Internet access service providers to offer such additional service to their customers. However, the limited computing and memory resources that residential gateways have by design make the implementation of new features working at wire-speed very challenging, particularly when complex operations such as parsing packets up to the application layer (a.k.a. Deep Packet Inspection or DPI) are involved. This is the case for many critical modern filtering applications, such as malware protection, corporate policy enforcement, parental control, advertisement block, that are based on inspection and filtering of Uniform Resource Locators (URLs). In fact, users access and exchange content mostly through mobile apps and web applications, both based on HTTP, which

uses URLs to identify data objects to be transferred.

Currently, the above URL filtering-based services are most often operated in web proxies [2] or in end-user devices (e.g., laptop, tablet, smartphone), as a mobile app [3] or a browser plugin [4]. None of these solutions can guarantee that all the outgoing traffic is analyzed and filtered; in fact, a user can switch to a different device, disable the filtering software or change the client network settings in order to bypass a web proxy. The residential gateway is the perfect spot where to implement services that require all the web page requests to be analyzed. This would require matching URLs against large, dynamic blacklists, which far exceeds the limited hardware capabilities of this category of devices. For example, an effective parental control service, which is a valuable offer to residential customers, is based on a very large database of URLs that cannot be stored in the limited memory of common residential gateways (usually in the order of tens of MB). An additional challenge comes from the fact that the database must be frequently updated. Last but not least, URL matching cannot be limited to the hostname, but the entire URL should be considered because the same web server can host both appropriate and inappropriate or malicious pages. Hence, looking up a URL within a huge list of blocked resources exceeds the processing capabilities of a residential gateway, especially if it must be done for live traffic, which implies that the additional introduced delay must be limited.

This paper presents *U-Filter*, an efficient solution to integrate a URL filtering service in a resource constrained device, such as a common residential gateway, leveraging a distributed architecture. A remote *policy server* in charge of keeping the URL database up-to-date provides a fast API that can be accessed through the network in order to establish if a request for a specific URL is allowed. It is reasonable that the above mentioned server is operated by a service provider (or the network service provider) and can rely on powerful hardware resources to serve multiple residential gateways with minimal response time. However, this architecture does not necessarily require the network service provider awareness and collaboration. The presented solution greatly alleviates the load on each

residential gateway, even though it must still perform a limited form of DPI on outgoing packets to extract the URL from every HTTP request, and afterwards query the server in order to determine the policy that must be applied. We adopt specific techniques to optimize this task and limit the latency introduced by the client-server interaction, striking a balance between the load they introduce and the limited resources available in residential gateways. Although the U-Filter design and the adopted optimizations are presented here in the context of policy enforcement on HTTP traffic, they offer a general solution for in-network policy enforcement suitable for a wide range of network protocols, thanks in particular to the decoupling of policy checking and enforcement phases, as detailed in Section 2.2.2.

This paper is organized as follows. Section 2 presents the architecture of U-Filter, describing the design principles that led to our solution and the optimizations used to provide real-time policy enforcement on resource-constrained devices. In section 3 we evaluate the proposed solution by discussing its limitations and analyzing the additional delay introduced by U-Filter. We validate U-Filter in Section 4 through various experiments showing the impact on the user experience. Section 5 presents the state of the art of HTTP-level policy enforcement and Section 6 concludes the paper with a discussion of future research directions.

2. Architecture and implementation

2.1. Operating principles

A typical deployment scenario of U-Filter is presented in Figure 1. A user surfing the web generates many HTTP requests that transit through her/his residential gateway. These requests are analyzed by U-Filter, which extracts the requested URL through a lightweight DPI algorithm. This allows to process line rate traffic with a small overhead for the residential gateway. Afterwards the HTTP request is released and can continue its journey towards the web server, while the URL is simultaneously sent to the policy server that provides the

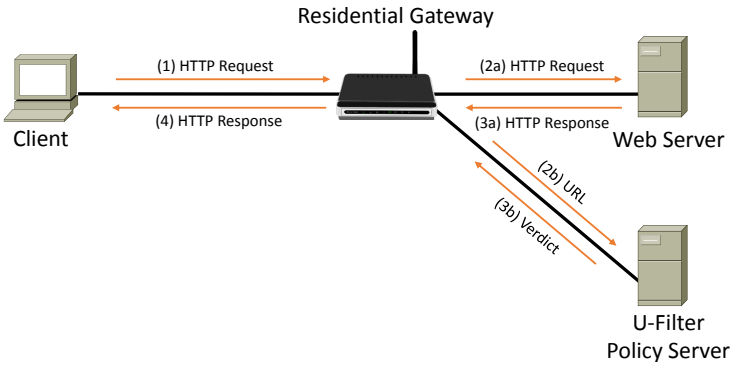


Figure 1: U-Filter workflow.

policy to enforce. This policy is enforced by U-Filter on the packet carrying the HTTP response by either blocking or allowing it. Thanks to the parallelization of the policy server and web server processing, this workflow greatly reduces the latency experienced by the user, making it comparable with the one that can be obtained with the same hardware without the service in place.

2.2. Architecture overview and design principles

Our prototype has been built around three objectives. First comes **flexibility**, as it is essential to be able to enforce effective protection to end users in a prompt response to newly discovered threats. Second is **efficiency** since the system is targeted to resource-constrained devices. Third, we took care of ensuring an excellent **user experience**, hence limiting the impact of the system in terms of possible additional latency when inspecting traffic to apply filtering policies. The above high-level objectives have translated in the following four design choices.

2.2.1. Three-tier processing architecture

As shown in Figure 2, U-Filter includes (*i*) an *online module*, which sits on the data plane of the router and is mainly in charge of identifying (and extracting) requested URLs from network traffic (more details in Section 2.5) and apply the policy decisions on the return traffic, (*ii*) an *offline module* that

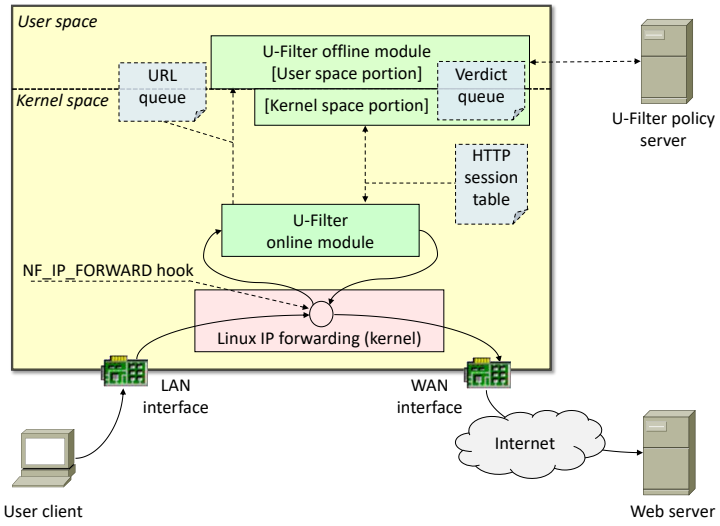


Figure 2: U-Filter architecture.

queries a remote policy server to know whether such URL should be allowed or not (described in Section 2.6), and (*iii*) a *remote server* that implements the complex protection logic and returns a boolean value with the result of the classification, i.e., if the corresponding HTTP session handled by the online module has to be allowed or the URL is malicious and the response has to be blocked. The first two modules are built with efficiency in mind, while the latter allows to achieve the required flexibility.

The U-Filter online module is inserted on the path that packets being forwarded by the residential gateway take through the system. It leverages a hook provided by the `netfilter` [5] framework, as detailed in Section 2.3, available in the mainline Linux kernel, to enable interaction with the IP forwarding function. To achieve high performance, the online module is executed in the kernel space; this allows to avoid expensive kernel-to-user context switching and enables sharing the required data structures with the rest of the kernel (e.g., direct access to privileged memory areas), hence minimizing communication overheads. In fact, by working in kernel space, the online module can implement a *zero-copy* approach, since the data structure containing the packet data is not copied in

the user space memory and is only referenced by the online module. On the other hand, the offline module is invoked a limited number of times compared to the online module because it operates only when a new URL is detected, but it requires more time to complete due to its interaction with the (remote) policy server. As a consequence, an asynchronous execution model is preferred for this module in order not to block the execution of the data path. This could be implemented as either a dedicated kernel thread or as a user-space process, which is the solution chosen in our implementation[1] because of the complexity of the tasks it executes and to avoid that any possible misbehavior (or bug) can be propagated to the kernel, hence affecting the overall operation of the residential gateway.

The policy server can be executed on a remote host (or on a cluster of hosts for performance reasons), as its only interaction with the rest of the system is through a query/response protocol. A single policy server can be queried by offline modules running on multiple (remotely distributed) residential gateways. In our implementation, this interaction has been implemented with the ad-hoc dedicated protocol detailed in Section 2.7, but other choices (e.g., REST web service) are surely possible.

2.2.2. Decoupling policy verification from HTTP operation

As introduced in Section 2.1, policy compliance is verified without holding outgoing packets on their ride towards the final destination. This solution makes the system more complicated but much more efficient. In fact, keeping the HTTP request on hold until the arrival of the response from the policy server would add additional delay to the HTTP communication, increasing the Round Trip Time (RTT) of the HTTP connection and hence affecting the user experience. Vice versa, the U-Filter offline module checks the requested URL with the policy server during the normal HTTP RTT. A temporary entry in an

[1] In fact, a small portion of the offline module has to be implemented anyway in the kernel space, as shown in Section 2.6.

HTTP session table is created by the online module in order to possibly hold a response from the web server received *before* the result of the compliance check arrives from policy server. While this allows packets to travel through the Internet also if they are part of a session that shall be stopped, the answer from the web server never reaches the user, effectively preventing possible unwanted data to reach the user's host.

2.2.3. Efficient memory usage

Efficient memory usage is a key problem because of (*i*) the limited amount of memory usually available in current residential gateways, and (*ii*) the bad effects in terms of CPU cache pollution when large memory structures (with sparse access patterns) are used. Several implementation choices have been adopted to ensure that memory is used efficiently. According to the best practice for kernel module development, all the memory used by the online U-Filter module is allocated at startup in order to avoid costly memory allocations at run-time, and the structures that are used for the communication between online and offline modules are shared (using the proper primitives for mapping memory between kernel and user space) for better memory efficiency. Furthermore, all the helper structures (detailed in Section 2.4) make use of contiguous memory areas in order to improve data locality and, as a result, CPU cache efficiency, except for the packets that may need to be held temporarily by U-Filter (while waiting for an answer from the policy server), which have been allocated by other portions of the kernel and therefore are not under our control. Finally, the usage of additional memory is kept at minimum: (*a*) the data structure dedicated to the session table defines a "default" behavior that avoids storing accepted sessions, and (*b*) the number of packets held by the router while waiting for the answer from the policy server is limited to, at most, *one per session*, hence further reducing memory requirements.

2.2.4. Per-packet operation

This is known to be much more efficient than per-TCP session processing while, at the same time, reducing the latency required to extract application level information (namely URLs). In fact, the former can be based directly on the very efficient packet processing primitives available in the Linux kernel through the `netfilter` framework, instead of requiring a full-blown HTTP proxy, whose complexity is so high to make a kernel implementation problematic. Therefore, an additional overhead is added for moving all packets from kernel to user space, where a proxy is usually located, and then back to kernel for their transmission on the output interface.

As a downside, working on individual packets makes the system less robust against malicious attacks such as HTTP requests whose URLs are split across packets (possibly deliberately sent out of order). Such attacks could be spotted by adding lightweight, packet-based ad-hoc anomaly detection algorithms [6, 7, 8], which is outside of the scope of this paper.

2.3. Netfilter

In order to gain access to live traffic, U-Filter leverages `netfilter` [5], a framework provided in the mainline Linux kernel that allows analyzing and modifying all the packets that are being received by the kernel. `netfilter` defines a set of hooks that correspond to different stages in the path packets take in the system. An application can register one or more callbacks linked to a specific hook; the corresponding callbacks are invoked whenever a packet passes through it. The callback receives a pointer to the system data structure containing the packet's data as a parameter, therefore it can read and modify the packet. Finally, the returned value instructs the system on whether the packet can continue its journey (`NF_ACCEPT`), or should be immediately dropped (`NF_DROP`), or should be diverted to a different (custom) processing pipeline (`NF_STOLEN`), which is useful if the decision about accepting/dropping the packet has to be postponed.

Figure 3 shows the possible paths taken by packets, together with the hooks

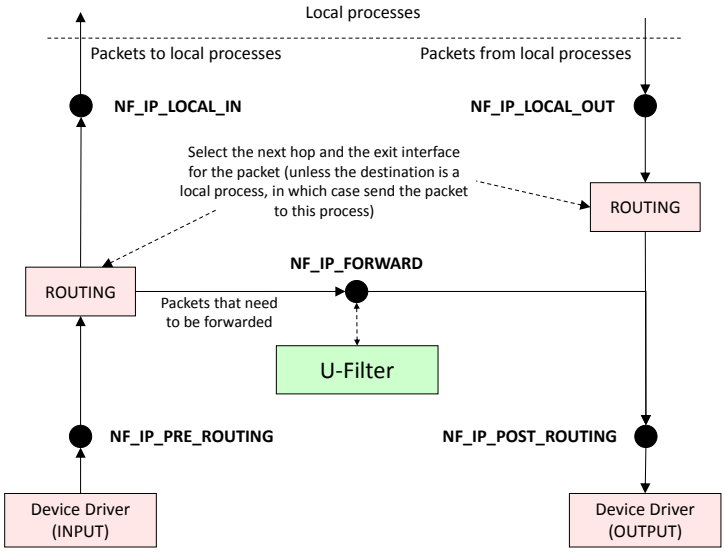


Figure 3: `netfilter` hooks chain and U-Filter.

that can be used to register callbacks. All the incoming packets are caught by the `NF_IP_PRE_ROUTING` hook, before being processed by the routing task; afterwards, packets addressed to the host itself are caught by the `NF_IP_LOCAL_IN` hook, while those traversing the host on their way toward the destination hit the `NF_IP_FORWARD` hook (where U-Filter is attached). The `NF_IP_LOCAL_OUT` hook catches packets sent by the host's local processes, while the `NF_IP_POST_ROUTING` hook catches all the outgoing packets, whether they are forwarded or locally generated.

2.4. Key data structures

The online and offline modules exchange data using three shared structures, as shown in Figure 2: (*i*) a hash map for the status of the policy for a given session, (*ii*) a queue for the URLs that have to be send to the policy server and (*iii*) a queue with the verdict received from the policy server. Each of the data structures is described in detail in the reminder of this section, while their usage will be discussed in the following sections.

The *HTTP session table* (shown in Figure 4) stores data regarding pending

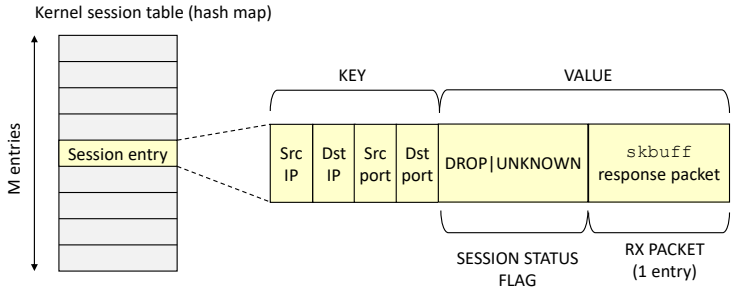


Figure 4: HTTP session table, shared between online and offline modules.

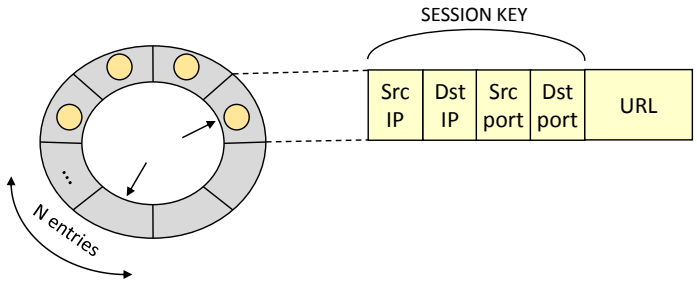


Figure 5: URL queue, shared between the online module and the offline module user space process.

sessions. An HTTP session is considered pending when the HTTP request has been received, but either the HTTP response from the web server or the decision from the policy server are yet to be received. The hash map implementing the HTTP session table is allocated in kernel space and is shared between the online and offline module because the former needs to know (when an HTTP response arrives) whether a decision for an URL has been received, while the latter needs to know, when the verdict is available, whether an HTTP response is already waiting. An entry in the HTTP session table can be deleted as soon as both the HTTP response and the verdict from the policy server have been received.

The *URL queue* (shown in Figure 5) is shared between the online module and the offline module user space process, while the *verdict queue* (shown in Figure 6) is shared between the kernel thread and the user space process of the offline module. The two queues are managed according to a FIFO policy and

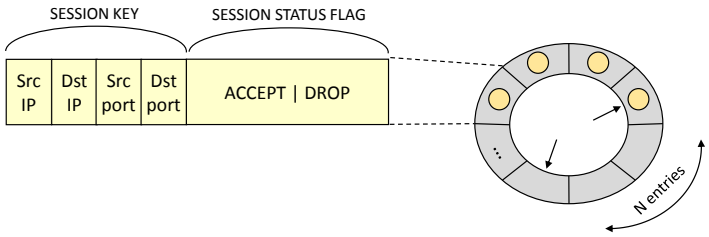


Figure 6: Verdict queue, shared between the offline module kernel thread and user space process.

the access to each queue is implemented with two pointers, pointing respectively at the first *free* and the first *full* slot.

To correlate data in different data structures, an entry always contains a key made by the 4 tuple identifying the TCP session (later referred as session ID):

$$(Source\ IP, Destination\ IP,\ Source\ TCP\ port,\ Destination\ TCP\ port)$$

The addresses are the ones present in the HTTP request and are inverted in the corresponding HTTP response.

An entry in the URL queue contains also the URL that should be checked with the policy server, while an entry in the verdict queue contains a session status flag that assumes either `ACCEPT` or `DROP`, according to the policy to enforce. The URL is stored in some pre-allocated memory whose size allows containing a full-length HTTP payload (i.e., 1460 bytes), in order to avoid memory allocations at run-time. On the other hand, an entry in the HTTP session table stores as value a session status flag and a void pointer to a packet (`skbuff` structure, allocated by the operating system). The use of this pointer is detailed in Section 2.5. Differently from the verdict queue, the session status flag in the HTTP session table can assume either `UNKNOWN` or `DROP`. In fact, entries corresponding to an `ACCEPT` policy are deleted as soon as the verdict is available in order to reduce the size of the hash table. Thus, in the HTTP session table the absence of an entry is considered as an `ACCEPT` policy.

As a further optimization to reduce the allocated memory, in our prototype the TCP session ID uses only the last byte of the source IP address, instead

of the entire 4 bytes address, with no impact on the system proper execution. This optimization is correct in our environment, since domestics LANs usually adopt a 24 bits subnet, therefore all the clients have the same value for the first 3 bytes of the IP address. In general this is not valid for every deployment, hence the optimization should be adapted to the specific addressing plan in use.

2.5. Online module

The online module sits on the data path by intercepting all the traffic forwarded by the router through a callback registered on the `NF_IP_FORWARD` `netfilter` hook[2]. As shown by the workflow depicted in Figure 7, most of the processing occurs when an HTTP *request* or *response* is detected. For each packet, the module first locates the beginning of the TCP payload and then checks if that packet can be considered the *first segment* of an HTTP request or response by matching the beginning of the TCP payload against a few simple text strings, namely an HTTP method (i.e., GET, POST, PUT, etc.) in case of a request or a version string (i.e., HTTP/1.0 or HTTP/1.1) in case of a response. This classification method is far more reliable than checking the transport-layer port number, as investigated in [9]. All other packets, namely HTTP packets that are not the first of the request/response message (hence, do not match the signature), as well as non-HTTP traffic, are left to continue their way as the online module returns `NF_ACCEPT` to `netfilter`. Notably, since *all* TCP packets containing a valid payload are matched against the signature, this algorithm is able to intercept *all* the HTTP requests/responses that are issued within a connection in HTTP 1.1 persistent mode, not only the first one, as well as within HTTP connections terminated on a non-standard TCP port. This algorithm could raise concerns about the cost of inspecting all packets, as general DPI techniques are normally demanding in terms of computing resources. However, our algorithm does not perform a full-blown DPI with full parsing of

[2]By choice, U-Filter does not apply policies to the packets that are received and generated by the router itself, e.g., for management purposes.

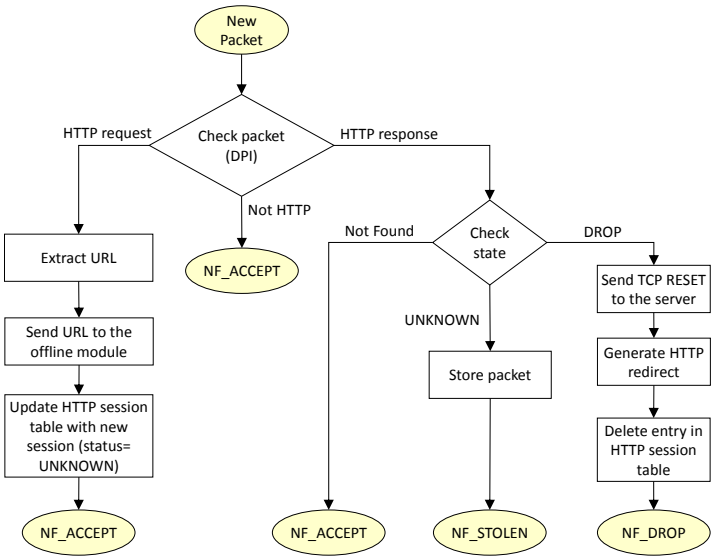


Figure 7: Summarized workflow of the online module.

all protocol headers and their fields. Instead, it performs a lightweight parsing to locate the beginning of the TCP payload and a *string checking* (instead of regular expressions) just on the *initial bytes* of the payload, which is a reasonable assumption that is discussed in Section 3.1. In fact, our experimental validation (Section 4.3, Figure 13) confirms that the online module does not introduce noticeable overhead in the traffic processing.

In case of an HTTP request, the URL is extracted and sent to the offline module by pushing a new entry in the (shared) URL queue (Figure 5), which includes the TCP session identifier to later match the verdict from the policy server with the corresponding HTTP session. A new entry is also created in the HTTP session table; as shown in Figure 4, it includes the TCP session identifier (as a key), a session status flag that is marked as `UNKNOWN`, and an additional field that is left empty. Afterwards the packet is allowed to be forwarded by returning `NF_ACCEPT` to `netfilter`.

When an HTTP response is received, the module checks the status in the HTTP session table and acts according to the three possible scenarios:

- The lookup is successful and the requested URL is forbidden (`DROP` in the session status flag). The HTTP response is dropped (i.e., a `NF_DROP` is returned to `netfilter`), and two new packets are generated: (*i*) a TCP RESET message sent to the web server to forcibly close the connection and (*ii*) an HTTP redirect message sent to the client in order to show the user a courtesy web page notifying that the requested web resource was blocked. Moreover the entry is removed by the HTTP session table.

- The lookup is successful but the system is still waiting for the policy server to respond (`UNKNOWN` in the session status flag). This occurs when the response from the web server arrives *before* the one from the policy server. In this case the HTTP response packet is put on hold by returning `NF_STOLEN` to `netfilter` and saved in the proper `skbuff` structure (shown in Figure 4) of the HTTP session table entry, waiting for the arrival of the answer from the policy server. This is the only case in which the user experiences an additional delay compared to a scenario where U-Filter is not deployed; a characterization of this delay will be provided in Section 3.3.

- The lookup is unsuccessful. Our algorithm interprets this condition as the URL being allowed, hence the HTTP response is forwarded to the client. Since in common URL filtering applications most URLs are not to be blocked, this design choice allows considerable space savings in the HTTP session table (Figure 4), as we avoid explicit entries for all the sessions that correspond to 'accepted' URLs.

Notably, the algorithm needs to hold (hence, store in the kernel session table) no more than *one* packet per HTTP session. In fact, even if other segments of the HTTP answer are in fact delivered to the destination, the TCP layer on the destination host cannot reconstruct the entire message because of the missing packet, which is the first segment of the HTTP response. This prevents the message to be actually delivered to the application (e.g., web browser) while

keeping at minimum the memory storage requirements in the residential gateway. However, this solution also causes the transmission of some duplicated packets, which we analyze in Section 4.2 and that are discarded by U-Filter since they are equal to the packet already on hold.

2.6. Offline module

As depicted in Figure 2, the offline module is split in two portions, the first one operating as a process in user space, while the other operates as a thread in kernel space. The former is in charge of the communication with the policy server, as shown in Figure 8, while the latter executes the workflow summarized in Figure 9.

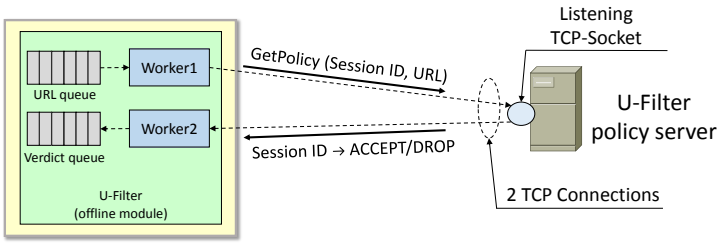


Figure 8: Offline module user space process.

The user space process retrieves URLs from the URL queue and sends them to the policy server, which provides decisions stating whether they are acceptable or to be blocked. These decisions are then pushed in the shared verdict queue, together with the same TCP session identifier that was stored in the corresponding URL queue entry.

The entries in the verdict queue are retrieved by the offline module thread in kernel space, which reads the enclosed decision. In case the resource is legitimate (the entry contains the `ACCEPT` flag), it checks whether a packet is stored in the HTTP session table entry corresponding to the TCP session key present in the verdict queue entry. This packet, if present, is injected back into the networking stack of the operating system, exactly in the same point of the `netfilter` chain where it had been stolen, so that the packet is processed by any other software

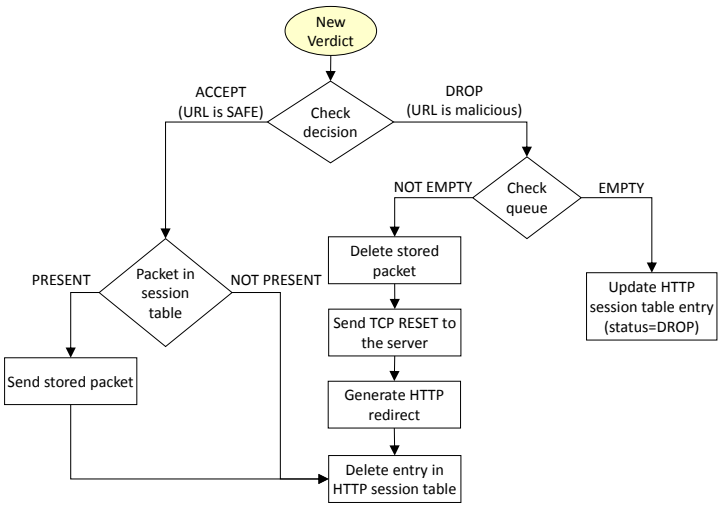


Figure 9: Summarized workflow of the offline module kernel thread.

relying on `netfilter` (e.g., NAT). The HTTP session table is then updated by deleting the entry since, as mentioned earlier, the absence of an entry is interpreted as an `ACCEPT` verdict. The `skbuff` structure containing the first packet of the HTTP response is stored in a memory location managed by the operating system, hence the offline module leverages the kernel space thread to access it.

In case the resource is not legitimate (the verdict queue entry contains the `DROP` flag), if no packet is found in the HTTP session table entry, the session status flag is updated to `DROP`, thus the online module will drop the response packet when it arrives. If a packet is already stored in the HTTP session queue entry, the offline module performs the same actions previously described for the online module in case of a `DROP` policy. Additionally the packet is dropped, so that the client cannot reassemble the HTTP response.

Additionally, the last N unauthorized URLs are cached in the offline module. Each URL is first looked up in the ad-hoc verdict cache and, in case of a hit, there is no need to interact with the policy server and redirection to the courtesy web page can be immediately implemented, thus reducing the overhead for the

module.

2.7. Communication with the policy server

The U-Filter offline module exploits two different parallel threads to interact with the policy server, each one using a distinct TCP connection as shown in Figure 8. The two threads establish the TCP channels when the system starts, hence enabling the offline module to send immediately a query to the policy server when needed, without the overhead (and the consequent latency) of the TCP handshake[3].

The offline module exploits these threads to implement an asynchronous communication with the policy server, separately processing the requests and the replies without any wait. The first thread cyclically collects every new entry present in the URL queue and sends the URL and the TCP session identifier to the policy server, which replies with a message on the second thread, using the second connection, containing the same Session ID and a single binary information (`ACCEPT`/`DROP`) that is used to push a new entry in the verdict queue. This solution allows to process as fast as possible both new entries in the URL queue and new replies from the policy server. The Session ID sent back and forth is used to correlate the requests with the replies, so that there is no need to share data between the two threads. Since the requests are sent sequentially, the policy server can adopt different techniques to efficiently parallelize the policy checking, such as spawning new threads without the necessity to open a dedicated TCP connection for each of them.

It is worth noting that most TCP implementations are designed to use the Nagle algorithm by default, in order to reduce the congestion of the network and increase bandwidth efficiency at the expense of latency [10]. This algorithm buffers application data until all the previously sent packets are acknowledged

[3]The messages sent to and received from the policy server are not intercepted by the callback of the online module, since they are addressed to the local host and do not cross the `NF_IP_FORWARD` hook, where the callback is registered.

or the data reach the Maximum Segment Size (MSS). In this way the probability of having small packets in the network (i.e. packets smaller than the MSS) is strongly reduced, thus limiting the overhead of TCP headers, allowing for a more efficient use of transmission links and reducing the burden on routers in terms of packets per second to be processed. This behavior is particularly harmful for U-Filter, since both the offline module and the policy server always send very small packets, that most of the time would be delayed up to one RTT. It is therefore crucial that the offline module and the policy server disable the Nagle algorithm (typically with the `TCP_NODELAY` socket option) when establishing the two connections.

3. Discussion

This section analyzes the proposed technique in terms of possible limitations (among the others, its applicability to encrypted traffic), and it performs a theoretical characterization of the delay that can be possibly added by U-Filter on real network traffic, which will be validated in the next section dedicated to experimental evaluation.

3.1. General limitations

The proposed solution has been designed with the aim of providing small delay and low overhead on resource-constrained residential gateways. This was traded for some limitations compared to more complex solutions adopting a full-stack HTTP proxy.

The matching process is meant to keep the number of string matching operations as small as possible, and surely it has to avoid to completely inspect the entire payload of all the packets in order to identify HTTP messages and extract URLs in a reasonable amount of time. Therefore, this solution does not handle correctly packets where the HTTP header is not at the beginning of a packet. This is not a relevant limitation since the problem arises only when HTTP

pipelining[4] is enabled, which is rarely the case in common browsers [11, 12]. The matching algorithm also cannot handle sessions where the header of the HTTP request spans multiple packets and the necessary fields (e.g., the *Host* field) are not on the first one. According to [13], less than the 5% of HTTP requests are bigger than the common 1500 byte Ethernet maximum transmission unit. Considering that large HTTP requests are often POST messages carrying a long payload, e.g., users submitting the content of a form to a web service[5], the possibility that the URL cannot be extracted from the first packet is presumably much smaller than this amount.

Moreover, various encapsulation techniques (e.g., GRE tunnels) are not supported by the presented version of the algorithm. These limitations can be avoided at the cost of additional complexity of the URL extraction procedure.

3.2. HTTPS

HTTPS uses data encryption to guarantee confidentiality, which makes traffic opaque to a possible observer. As a result, any in-network service requiring visibility into application layer content, such as U-Filter, becomes ineffective. Several studies [14, 15, 16] have addressed the problem of HTTPS traffic processing in middleboxes, which shows that this is a general open problem, not specific of U-Filter. As a sample general solution, [14] proposes an evolution of HTTPS that supports the operation of trusted middleboxes while retaining the security properties of HTTPS. We leave as future work the analysis of the interaction of U-Filter with such solutions.

We can envision a number of ways to enable U-Filter to operate (possibly

[4]HTTP pipelining allows a client to send multiple HTTP requests on a single TCP connection without waiting for the corresponding responses. It requires support in both the client and the server.

[5]It is worth noting that this case falls outside the scope of U-Filter, as the apparent URL submitted in an HTTP POST request contains, in fact, user data. As a consequence, this would require a more sophisticated filtering mechanism based on a *content* inspection, not just *URL* inspection.

with limited capabilities) on HTTPS traffic. A first option is to deploy a *trusted proxy* [17], such as the one presented in [18], at the cost of a significant processing overhead, which inevitably limits the performance on a resource constrained device like a residential gateway, as shown in Section 4.4 with respect to a similar solution.

Secondly, U-Filter can be extended to inspect unencrypted messages exchanged during the TLS session establishment, extract the domain name (from the fields Common Name, Subject Alternative Name or Server Name Indication), and enforce a policy according to the extracted value. With this solution it is possible to block only an entire domain, not just a single resource. It is worth noticing that a client can resume a previously established TLS connection with a web server by sending a past TLS session ID in the first message, which results in an abbreviated handshake without the exchange of the server domain name. Thus, if the initial connection was not inspected (e.g., because it was performed on a different, unprotected network), it is not possible to discover the server domain name by looking only at unencrypted data. Although this happens only in a quite uncommon network setup, it is to be kept in mind that the solution is not bullet proof.

As studied by [19], the cost of the security provided by HTTPS is non-negligible in particular in case of mobile devices and smart objects. In addition, there are a number of applications for which confidentiality is not strictly required, for which their users may not willing to pay the additional cost of the encryption. Therefore a significant fraction of HTTP traffic is expected to remain unencrypted in the near future. Although we leave to future work the architectural and implementation details of a solution to support HTTPS traffic, we envision U-Filter as a low-cost solution for URL filtering on the fast path of HTTP traffic, while HTTPS traffic can be steered toward a slower path, where a trusted proxy is used to provide the same level of policy enforcement.

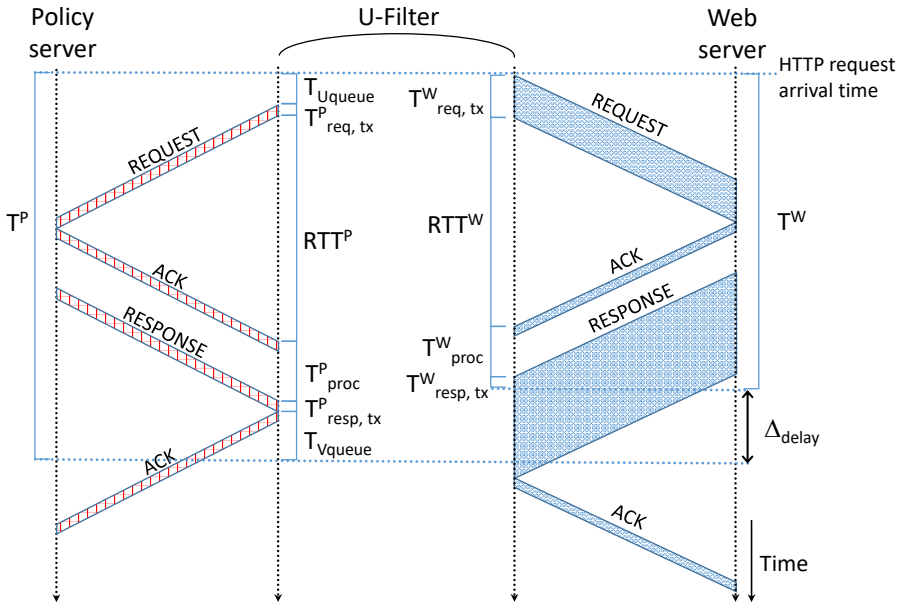


Figure 10: Delay characterization.

3.3. Delay characterization

In this section we analyze the additional delay introduced by U-Filter to identify the components that can be relevant and must be evaluated to quantify the impact on the user experience.

Specifically, the delay experienced by the end user when requesting a web page depends on: (i) the time for having a verdict from the policy server T^P, (ii) the time until the first packet of the response from the webserver is received T^W, (iii) the difference between (i) and (ii) Δ_{delay}, as detailed in Figure 10. The latency in the communication from the client to the residential gateway is not relevant in this context since it is not affected by the presence of U-Filter.

Let's first characterize T^P. When U-Filter receives the first packet of an HTTP request, the online module extracts the URL, pushes a new entry in the URL queue and sends the HTTP request forward. The entry spends a time T_{Uqueue} in the URL queue, until it is extracted by the offline module and sent to the policy server, with a time $T^P_{req,tx}$ required to transmit the bits on the

channel. The verdict is available to the offline module after a Round-Trip Time RTT^P, a time T^P_{proc} required by the policy server to check its database and choose a verdict, and a time $T^P_{resp,tx}$ needed to transmit the response into the channel. At this point, the verdict is stored as a new entry in the verdict queue. An additional queuing time T_{Vqueue} lapses before the entry is retrieved by the offline module kernel thread and the proper action is performed to unlock the response. As a result, the total delay introduced by the policy checking process is equal to:

$$T^P = T_{Uqueue} + T^P_{req,tx} + RTT^P + T^P_{proc} + T^P_{resp,tx} + T_{Vqueue} \tag{1}$$

Moving now to the characterization of T^W, the time required to receive the first packet of the HTTP response from the web server is given by:

$$T^W = T^W_{req,tx} + RTT^W + T^W_{proc} + T^W_{resp,tx} \tag{2}$$

where:

- $T^W_{req,tx}$ is the HTTP request transmission time;
- RTT^W is the Round-Trip Time with the web server;
- T^W_{proc} is the time taken by the web server to provide the HTTP response (fetch a file, execute server side computation, query a database, etc.);
- $T^W_{resp,tx}$ is the time needed to transmit the **first packet** of the HTTP response.

The interval:

$$\Delta_{delay} = T^P - T^W \tag{3}$$

when positive, is the delay that U-Filter adds to any HTTP request. Experimentally, we observed that T_{Uqueue} and T_{Vqueue} are negligible, since the two consumer tasks are rather fast. Moreover, $T^P_{req,tx}$ is always less than $T^W_{req,tx}$,

since the request to the policy server contains only a small subset of the data contained in the HTTP request. Similarly, $T^P_{resp,tx}$ is always less than $T^W_{resp,tx}$, since the policy response packet is very small (it consists only of the session ID and a binary flag). Consequently, the most significant components of the U-Filter delay are the Round-Trip Times and processing times.

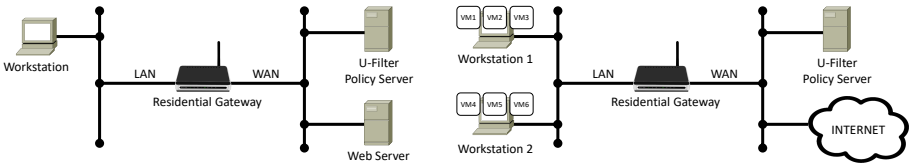
In case Δ_{delay} is negative, the user experience is completely unaffected by the presence of U-Filter. Even when Δ_{delay} is positive, though, thanks to the parallelization described in Section 2.7, the overall delay in a web page load time is not noticeable if the distance and the processing time of the policy server T^P_{proc} are comparable with the ones of common web servers, as shown in Section 4.

4. Experimental validation

In order to validate the proposed solution we conducted a broad range of experiments. Specifically our goal has been to study the interaction between the presented algorithm and TCP, as well as the conditions in which a web page load time is increased, quantifying to what extent the user experience is affected.

4.1. Testbed setup

We deployed U-Filter on a commercial low-cost residential gateway, a TP-Link Archer C7 (single core MIPS32 CPU clocked at 720MHz, 16MB Flash, 128MB RAM) running *OpenWrt* 12.09 [20] with the version 3.3 of the Linux kernel. OpenWrt is an open source operating system specifically optimized for the execution on resource constrained residential gateways. As shown in Figure 11, multiple workstations (whose number and setup varies according to the specific test) acting as clients are connected on a Gigabit Ethernet LAN representing the "domestic side" of the residential gateway. Another 1 Gbps interface ("WAN side") hosts the policy server and the traffic sink of our experiments, which is represented by a web server during TCP interaction and throughput experiments or a vanilla Internet connectivity when evaluating browsing experience. All the workstations and the servers are equipped with an Intel Core



(a) Testbed to analyze the interaction with TCP and to evaluate the maximum throughput.

(b) Testbed to evaluate the browsing experience.

Figure 11: Testbed setup.

i7-4770 CPU and 32GB of main memory in order to guarantee not to become the bottleneck.

Since a production-grade policy server is not in the scope of this work, we use a policy server that gives always a positive verdict, with a customizable delay in order to simulate the processing time. Moreover, in the policy server we use *Linux Traffic Control* (*tc*) to add a custom delay to any outgoing packet in order to simulate various network RTTs.

To generate single HTTP requests we use `curl` and `ab` [21], while for real-life simulations we start multiple VMs on the workstations to emulate multiple end-users. Each VM runs an instance of *WebTrafficGenerator*[6], an automation tool that can drive a web browser to replay a user browsing history. For every entry in the provided browsing history, the browser loads a complete web page (i.e. retrieving the web page with all the associated resources such as images, javascript files, etc.)[7]. In this respect, *WebTrafficGenerator* can also issue HTTPS requests, which happens when a page, appearing in HTTP in the browsing history, includes content that has to be retrieved using an encrypted connection. The time between multiple web page requests, a.k.a. the *Thinking*

[6]https://github.com/netgroup-polito/WebTrafficGenerator

[7]The community have not yet reached a consensus on when a web page should be considered completely loaded. Particularly, *WebTrafficGenerator* considers a page complete when the javascript "`onload`" event is fired on the "`body`" HTML tag.

Time, is randomly selected using a random variable with the same statistical distribution as the actual thinking time of the user as measured from his/her browsing history. A realistic thinking time is required not only to simulate a real user behavior, but also to avoid that web services (e.g. Google) recognize that the client is an automaton and thus provide a different response web page with the intent of testing whether or not the user is human. In the event that a new request must start before the previous web page is completely loaded, the tool creates a different browser window, in order to load multiple web pages in parallel (which simulates multi-tabbing).

4.2. Interaction with TCP

This section shows how the TCP algorithm reacts when one specific packet (the first packet of an HTTP response) is repeatedly lost on its way to the destination, for a certain amount of time. The aim of this analysis is to show that U-Filter has been designed taking into mind the peculiar characteristics of the TCP protocol, hence our algorithm that possibly delays the first packet of the HTTP response does not cause additional delay in the TCP data exchange.

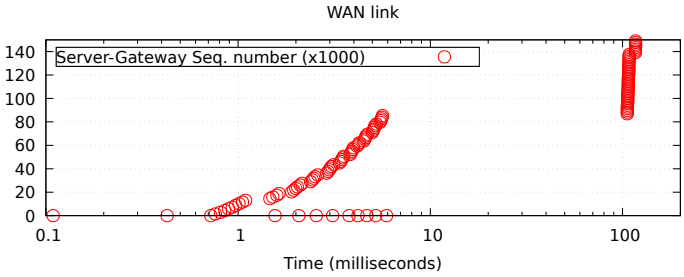
To reduce external interferences, in this test we use a web server directly connected to the WAN interface of the gateway (as shown in Figure 11a) running the *Apache HTTP Server 2.4.7*; T^W measured in this setup is less than 1 ms, thus we can consider $\Delta_{delay} = T^P$. Moreover, in this test the Linux Traffic Control (`tc`) in the policy server is disabled, hence the RTT is negligible and we can consider $T^P = T^P_{proc}$. A client workstation runs `curl` to request a 512 KB web page stored on the webserver. The gateway executes U-Filter with a fixed $T^P_{proc} \approx 100$ ms delay in the policy server response. As detailed in Section 2.5 and 2.6, only the first packet of any HTTP response is buffered by U-Filter. In the scenario created for these experiments, such packet is eventually forwarded to the client about 100 ms after the HTTP `GET` request traverses the residential gateway. All subsequent packets are forwarded correctly. We capture the traffic on both the LAN and WAN links of the residential gateway and extract the sequence numbers (SEQ) of the TCP segments from the web server to the client

and the acknowledgment numbers (ACK) of the ones from the client to the webserver, together with their timestamp. The resulting data are presented in Figure 12 (the SEQ and ACK numbers are relative).

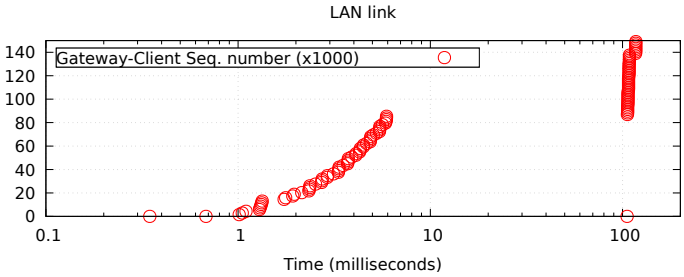
This experiment enables us to observe how a TCP connection progresses during the U-Filter operation. The presented results show that, while the first TCP segment of the HTTP response is blocked, the server TCP endpoint sends the subsequent segments as well as duplicates of the first segment (visible only on the WAN side, in Figure 12a), until the TCP window is full. As expected, the TCP receiver repeatedly acknowledges the segment arrived before the one missing (Figure 12c); specifically one ACK is sent for each of the subsequent segments received out of sequence. All the modern TCP implementations include the *TCP selective acknowledgment (SACK) option* [22] in the duplicated ACK, which is used to selectively acknowledge correctly received segments logically following the missing one(s). Thanks to the selective acknowledgments, these segments are not re-transmitted, as it happens for the blocked segment, as the traditional Go-Back-N algorithm would require. When the blocked packet is released (after 100 ms in our experiment, as shown in Figure 12b) and properly delivered, all the previously received segments are cumulatively acknowledged and the transmission can continue from a new segment (Figure 12c).

Abiding by *TCP Fast retransmit* [23] algorithm, the web server re-sends the blocked segment for every 3 duplicated acknowledgments. These re-transmitted segments are the only overhead induced by U-Filter. In our test these duplicates amount to 12.8% of the packets sent by the server during Δ_{delay}, and half that number if we consider *all* the packets transmitted during the same interval; however, considering the entire lifespan of the TCP connection, this overhead accounts (in average) no more than 1.6% of all the packets, which can be considered negligible.

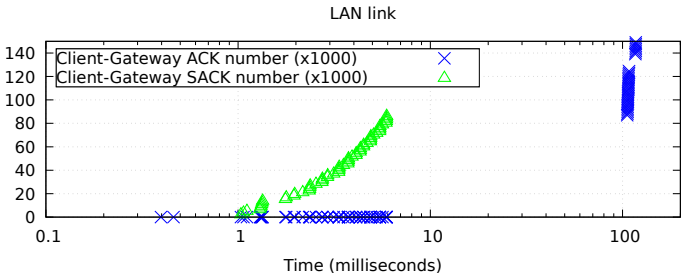
From the point of view of the users' experience, selective acknowledgments are particularly beneficial because, even if the policy server replies after the web server (i.e. Δ_{delay} is positive), the actual delay perceived by the user is smaller than Δ_{delay} because several TCP segments are correctly received during the



(a) Response packets timing on the WAN link



(b) Response packets timing on the LAN link



(c) Acknowledgement packets timing on the LAN link

Figure 12: Progress of a TCP session.

Δ_{delay} interval and are ready to be used to render the web page as soon as the missing segment is delivered.

4.3. Browsing experience

This section presents the results of several tests executed in a realistic scenario to show how much a real user browsing experience is affected by U-Filter. Using the testbed in Figure 11b, we launched *WebTrafficGenerator* in 6 VMs (running on 2 workstations) in order to simulate 6 users simultaneously browsing the Internet. This number of concurrent users is reasonable for a residential gateway. Moreover, with a large number of users, the browsing experience would be limited by the network speed. As expected, the latency of the policy server proved to be the parameter that has the greater impact on the user-perceived performance of U-Filter.

In every test, a single VM browses 600 web pages collected from the browsing histories of 30 anonymous users (we consider only web pages downloaded using HTTP, since those using HTTPS are irrelevant for U-Filter). In order to use realistic values for the policy server processing time and RTT, we analyzed several traffic traces captured using Tstat [24] during 24 hours in 4 different points of presence (POPs) of an Internet Service Provider and extracted the median and 90^{th} percentile values for the RTT of HTTP requests and processing time of web servers. Tstat infers the RTT from the POP to an endpoint by measuring the inter-arrival time of a packet and its acknowledgment and infers a web server processing time by measuring the interval between the arrival of the acknowledgment for the request and the arrival of the first response packet. In fact, a host's operating system usually sends a TCP ACK as soon as a packet is received.

Table 1 shows the statistical values for the RTTs from a client to the POP and from a client to the destination server, supposedly in a data center (DC). We use these values in our tests to simulate the RTT in the case that the policy server is either in the POP or in a remote data center. Additionally Table 2 shows the statistical values of the processing time for web servers. These values

Table 1: Inferred RTT values with the policy server in different locations (RTT^P).

Location	**Type of measure**	**RTT**
POP	Median	25 ms
	90^{th} percentile	100 ms
Data Center (DC)	Median	45 ms
	90^{th} percentile	200 ms

Table 2: Inferred policy server latency values (T^P_{proc}).

Type of measure	**Latency**
Median	2 ms
90^{th} percentile	80 ms

are used to simulate the processing time of the policy server: since the operations performed are somewhat similar (parsing of a request, look up in a database, preparation of a response), we assume the complexity to be comparable with (or even lower than) the one of any web server.

At the end of a test, *WebTrafficGenerator* provides a file containing a summary of various aspects of every request. Among the provided values, we are interested in the **complete page** load time (the time needed to load the web page with all its resources, such as pictures, libraries, etc.) and the timings of the **individual HTTP requests** issued to get the main HTML page and the associated resources.

4.3.1. Individual HTTP requests

The timing of an HTTP request is the sum of multiple components, such as the queuing time, the DNS resolution time, the connection setup time, etc. The only component that can be affected by U-Filter is the time spent waiting

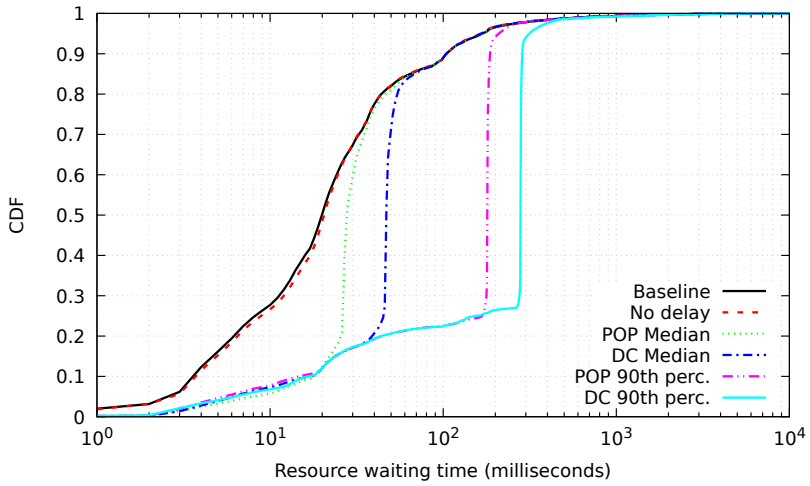


Figure 13: Waiting time for a single HTTP resource - Cumulative distribution function.

for a response from the server (*waiting time*), equal to $\max\{T^P, T^W\}$, if the RTT between the client and the gateway is negligible. Figure 13 shows the cumulative distribution of the waiting time for HTTP requests with different values of RTT and processing time (latency) for the policy server, together with the baseline (i.e., the latency without U-Filter) and the case in which the policy server immediately provides verdicts (in which case the delay T^P is negligible), as if U-Filter and the policy server are on the same LAN.

These results show that U-Filter adds a negligible delay if the policy server provides an immediate response, therefore proving our claim that the online module does not introduce noticeable overhead in the traffic processing. On the other hand, when the policy server response is received after a certain amount of time, the cumulative distribution is shifted toward that value, since all the HTTP responses that arrived earlier are delayed by U-Filter. In summary, the impact of U-Filter on the single resource loading time is highly dependent on the distance from the policy server and its processing time.

Considering only the worst case (i.e., the 90^{th} percentile of the processing time and RTT with the policy server in a data center), we show in Figure 14 the waiting time for each requested HTTP resource, with and without U-Filter.

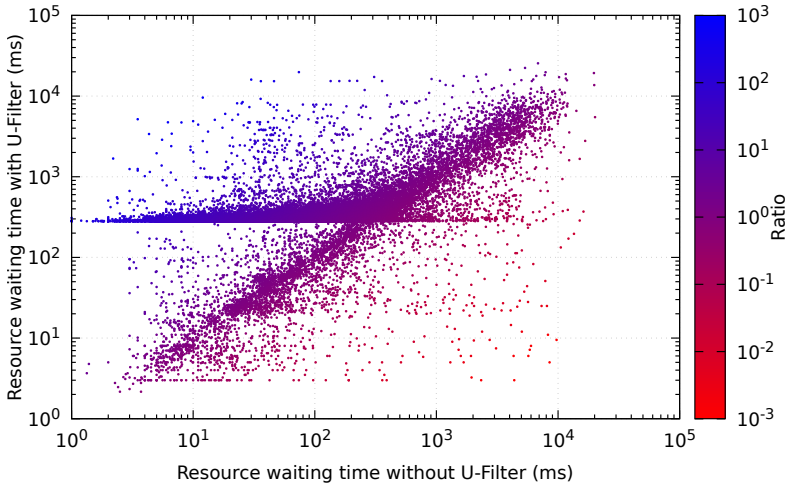


Figure 14: Resource waiting time considering the 90^{th} percentile of the processing time and RTT with the policy server in a data center.

The figure shows a cluster of requests on the horizontal line corresponding to the delay T^P, supporting the conclusion that this delay highly influences the loading time of a single resources.

Both figures show that, even with U-Filter, some resources are received before the policy server delay ($T^P \approx RTT^P + T^P_{proc}$). This happens because some resources are retrieved through HTTPS, even if the main HTML page is on HTTP, therefore they do not experience the policy server delay.

4.3.2. Complete pages

Figure 15 shows the cumulative distribution function of the complete web page load time, while Figure 16 shows for every requested URL the relation between the complete page loading time with and without U-Filter, in the worst conditions (policy server in the data center, 90^{th} percentile values for RTT and latency). These results show that the impact caused by the presence of U-Filter is not noticeable, therefore we can assert that the overall page loading time is not affected by U-Filter and also the browsing experience is unaltered.

This is justified by the fact that multiple resources are requested **in parallel**

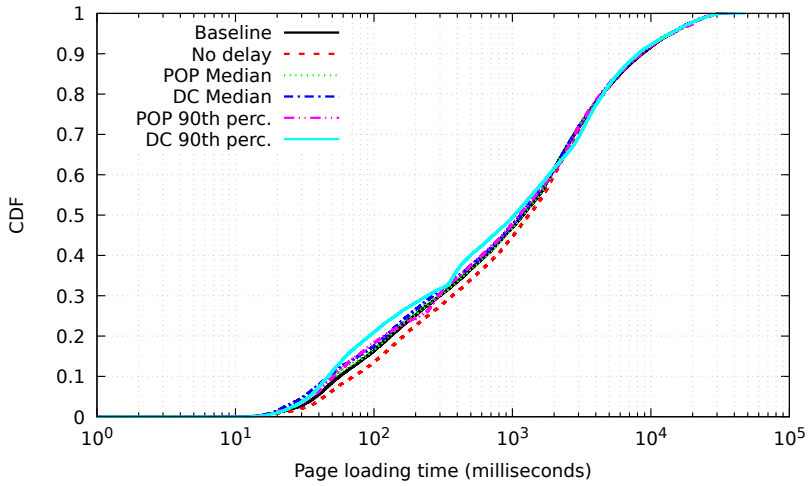


Figure 15: Complete page loading time cumulative distribution.

by the browsers, hence the policy server processes all the requests concurrently. As a result, the increase in the overall time for loading the complete web page is not dependent on the number of resources and is, in any case, approximately equal to a single policy server delay T^P. Since the time needed to receive, parse and render the main HTML web page and all its resources is usually an order of magnitude greater than the policy server delay, the added latency (and the impact of U-Filter on the browsing experience) is in effect negligible.

4.4. Residential gateway aggregated throughput

In this section we evaluate the overhead introduced by U-Filter by comparing the average aggregated throughput of the residential gateway in 3 scenarios: (*i*) without a URL filtering service in place, (*ii*) with U-Filter and (*iii*) with *Tinyproxy* [25], a URL filtering solution for OpenWrt based on a lightweight HTTP proxy that intercepts and analyzes all the outgoing web traffic and can operate in either explicit or transparent (a.k.a. man-in-the-middle) mode. These experiments assess the impact of U-Filter with respect to the maximum forwarding capabilities of the residential gateway, which is basically limited by the CPU consumption of the on-board software.

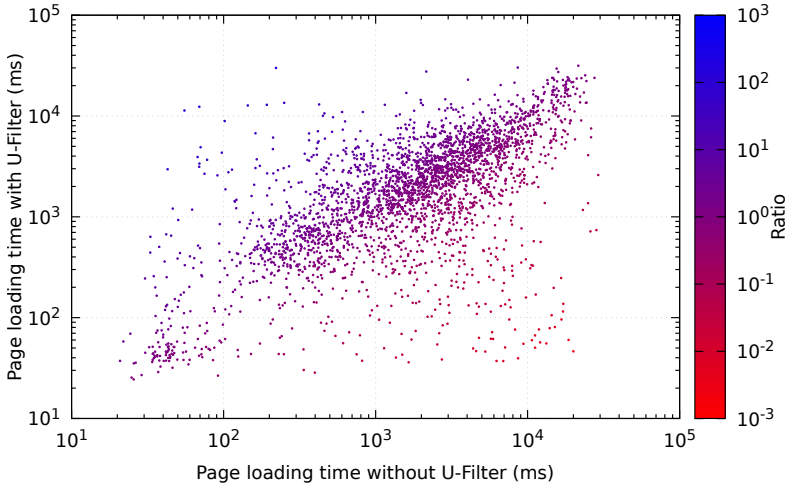


Figure 16: Complete page loading time considering the 90^{th} percentile policy server processing time with the policy server in a data center.

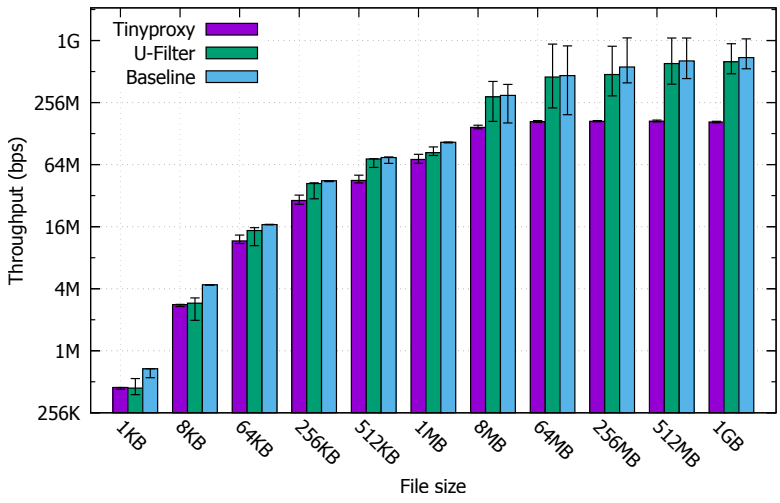


Figure 17: Application-level throughput when downloading files of different sizes.

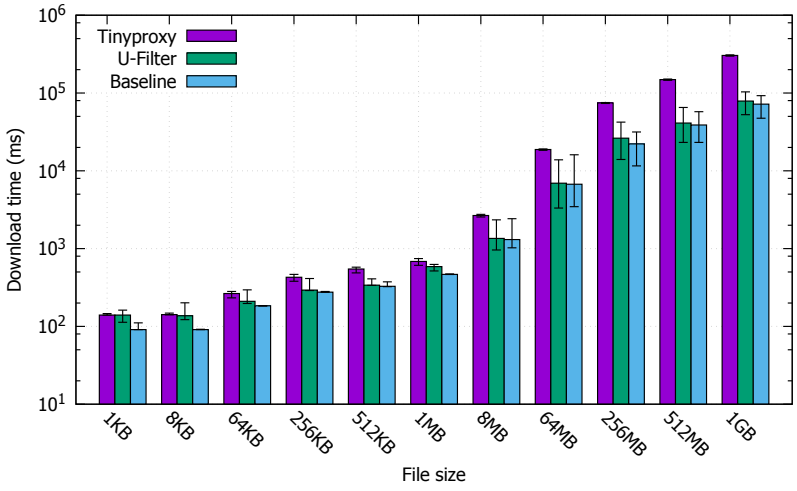


Figure 18: Download time when requesting files of different sizes.

These experiments employ the testbed setup depicted in Figure 11a; the policy server is configured to simulate a deployment in a data center with the median processing time and RTT, while the web server has the same RTT. The client workstation uses `ab` to request files of different sizes from the web server; each file is requested 100 times. As suggested by the HTTP/1.1 standard [26] with respect to persistent HTTP connections, each client issues two concurrent requests toward the server. The goal of this experiment is to evaluate how much packet inspection and policy checking in the residential gateway affects the download speed and the latency. We show in Figure 17 the minimum, maximum and average application-level throughput for the 3 scenarios, while in Figure 18 we show the time needed to download the entire file.

These results show that the throughput and the download speed reached with U-Filter are higher than with Tinyproxy for files larger than 8 KB, while for small files the two solutions show the same level of performance. In fact, with very small files, we experience an additional small delay with U-Filter, compared to the baseline. We ascribe this delay to the time needed for the context switch between the online and offline module, given that the residential gateway has a single core. This delay is negligible for larger files, for which

U-Filter provides almost the same performance reached without the filtering service in place. We expect that a residential gateway with at least a dual core processor would not experience this delay, therefore U-Filter would provide the same level of performance as the baseline. However, even with a single core gateway, the impact of U-Filter on the download time is only 3% with large files and never exceeds 54%, while Tinyproxy has an overhead ranging from 44% to a remarkable 322%. As an example, the download of a 1 GB file requires approximately 1 minute and 12 seconds without a filtering service, 6 seconds longer with U-Filter and more than 5 minutes with Tinyproxy.

It is worth mentioning that U-Filter can easily implement a whitelist containing the IP addresses of trusted devices whose traffic should not be filtered. This is a useful feature that allows to avoid the additional delay for delay-sensitive clients.

4.5. Memory footprint

Given the limitations in terms of available memory in current residential gateways, we extracted the number of pending entries in the HTTP session table every time a new HTTP request was received and plotted the resulting probability distribution in Figure 19 in order to assess the impact of U-Filter in terms of memory consumption. The observed values confirm the small memory footprint of U-Filter: even in the worst case, the number of pending entries are always less than a hundred. In the case in which every entry stores a packet (usually 1518 bytes at most), together with IP addresses (8 bytes), TCP ports (4 bytes) and a binary session flag, the HTTP session table requires less than 200 KB of main memory, a value far below the memory size of low-end residential gateways (usually in the order of at least tens of MB).

5. Related work

Currently several solutions for filtering traffic based on URLs are available commercially or as open source packages, often used as parental control or ad

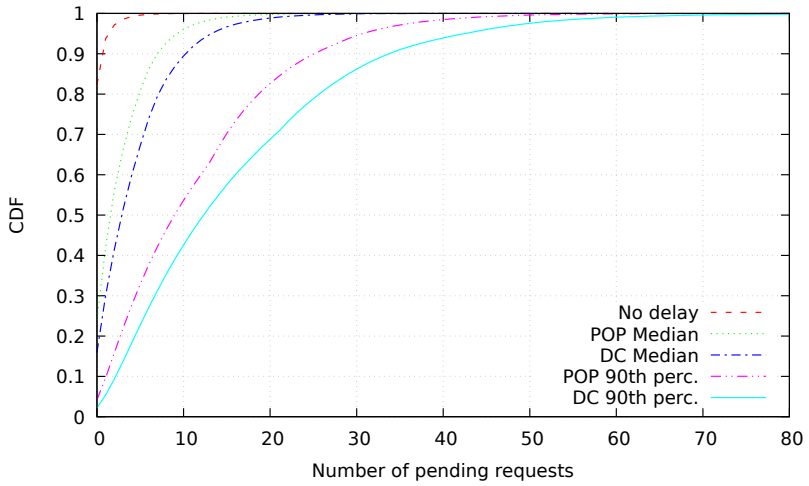


Figure 19: U-Filter load.

block. Many are based on software executing on the client machine to control outgoing traffic. Among them, it is worth mentioning *k9 Web Protection* [27], a powerful free software for URL filtering that comes with a large database of URL categorization data. New websites are categorized in real-time and their information published on a server that is used to update the local database. This software needs to be installed on any device that must be protected and is tuned to run on common PC hardware.

Among existing parental control solutions that do not require execution of a software agent on clients, some are based on applying the filtering policing in the DNS server [28]. While this is a low complexity and efficient solution that enables achieving high performance, it is not effective as it can be easily bypassed choosing a different DNS server. Moreover, filtering is based on server domain names rather than URLs, as required when the same server or name domain can deliver both appropriate and inappropriate content, such as in case of public services like `facebook.com`.

As an alternative approach, filtering policies can be applied by network appliances on the path of the protected client traffic. *Blue Coat WebFilter* [29] is a sophisticated URL filtering solution that runs on business level network appli-

ances and provides policy enforcement on web traffic, blocking malware downloads and web threats. *WebFilter* combines URL filtering and anti-malware technologies, exploiting an engine with a local rule database continuously updated from a remote master database. The engine detects hidden malware and provides reputation and web content categorization based on input from actual users.

None of the above-mentioned solutions is designed to run on resource-constrained devices, such as a typical residential gateway, which would not ensure acceptable performance when executing computationally intense tasks. Among the efforts to integrate web filtering service in low-end residential gateways, the ones related to the OpenWrt platform are noteworthy, such as Tinyproxy [25]. Tinyproxy can filter HTTP requests checking their URL against a list of regular expressions contained in a local file, which may be rather big and needs to be frequently updated. A similar technology has been proposed in [18], where an access gateway performs mobile app policy enforcement deploying a transparent HTTPS proxy to gain access to *encrypted* traffic, extract relevant field values, and pass them to an external policy-checking module. However, deployment of an HTTP proxy is critical on resource-constrained devices since it must terminate all the TCP connections, pair them with new TCP connections with the remote endpoint, parse every packet, identify and extract patterns of interest, and match them against a large blacklist. Therefore it becomes easily a bottleneck with high traffic loads, thus impacting user experience.

The work presented in [30] represents an attempt to perform efficient HTTP traffic filtering in *OpenWrt*. The authors propose a two-tier architecture, with a kernel module that intercepts and analyzes HTTP traffic and a user-space process in charge of policy compliance checking. The computational load of the user space module, that performs string matching on URLs, grows with the length of the list of rules, and so does the introduced delay. Consequently, when this approach is implemented on a residential gateway with limited resources, only short lists can be supported without user experience degradation, thus limiting the effectiveness of the policy enforcement system. Moreover, the

proposed architecture makes it difficult for a trusted third-party to push real-time updates to the local database in order to ensure prompt detection of newly discovered threats. Finally, the URL analysis is performed by each edge systems in isolation, hence excluding the possibility of a (centralized) cross-correlation mechanism that identifies new threats by analyzing URLs requested from different sources.

Traffic processing in residential gateways has been proposed also in the context of Network Function Virtualization (NFV) [31, 32]. An existing NFV infrastructure can employ residential gateways to deploy lightweight Native Network Functions [33] or eBPF data plane programs [34], in order to provide delay-sensitive services to the user, while computation intensive services are hosted in the data center of the service operator. This solution offers flexibility in the type and number of network services that can be provided and represents an interesting target platform for the deployment of U-Filter.

6. Conclusions

This paper presents U-Filter, a distributed system for efficient HTTP traffic filtering in resource-constrained residential gateways. Leveraging an external policy server and an intelligent combination of kernel and user space processing (and a careful implementation), U-Filter is able to inspect the URL in every HTTP request and block unwanted web pages with a very small memory footprint and processing overhead. This makes U-Filter appropriate for the deployment on resource-constrained devices and also reduces at a minimum the additional delay introduced on page download, which leaves the overall browsing experience of the user practically unaltered.

Since U-Filter operates on a packet-by-packet basis, it assumes that the entire HTTP header is on the same packet. This makes URL extraction easier and avoids to have to store additional information to correlate subsequent packets. Since the maximum size of an IP packet is usually 1500 bytes, this does not represent a problem in a real scenario, as confirmed by [13].

The policy server, where multiple mechanisms and optimizations can be implemented, was purposely kept outside of the scope of this work as it involves a completely different set of challenges and solutions. Similarly, we did not address how providing additional information to the residential gateway can increase its efficiency in caching verdicts, thus reducing the number of interrogations. The study of such improvements is left to future work. Other future directions will involve evaluating the performance improvement achievable by deploying U-Filter on real-time linux kernels [35, 36], as well as investigating the benefits that stem from deploying U-Filter in novel and currently strategic fields such as IoT [37] and Big Data [38].

Acknowledgment

We wish to thank Martino Trevisan and Marco Mellia for the statistics on real traffic traces and their initial implementation of the *WebTrafficGenerator*, and Pierpaolo Giacomin for the inspiring discussions and comments. We also thank the anonymous reviewers for their valuable feedback that helped improve the quality of this paper.

References

[1] Google Fiber.
URL `https://fiber.google.com`

[2] DansGuardian.
URL `http://dansguardian.org`

[3] Mobile Fence - Parental Control.
URL `http://www.mobilefence.com`

[4] Cloudacl WebFilter.
URL `http://www.cloudacl.com`

[5] The netfilter.org project.
URL `http://netfilter.org/`

[6] M. V. Mahoney, Network traffic anomaly detection based on packet bytes, in: Proceedings of the 2003 ACM Symposium on Applied Computing, SAC '03, ACM, New York, NY, USA, 2003, pp. 346–350. doi:10.1145/952532.952601.
URL http://doi.acm.org/10.1145/952532.952601

[7] S. Longchupole, N. Maneerat, R. Varakulsiripunth, Anomaly detection through packet header data, in: Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference on, IEEE, 2009, pp. 1–4.

[8] S. S. Kim, A. L. N. Reddy, Statistical techniques for detecting traffic anomalies through packet header data, IEEE/ACM Trans. Netw. 16 (3) (2008) 562–575. doi:10.1109/TNET.2007.902685.
URL http://dx.doi.org/10.1109/TNET.2007.902685

[9] A. W. Moore, K. Papagiannaki, Toward the accurate identification of network applications, in: Passive and Active Network Measurement, Springer, 2005, pp. 41–54.

[10] W. R. Stevens, UNIX Network Programming: Networking APIs, Vol. 1, Prentice-Hall, Inc., 1997.

[11] F. Schneider, B. Ager, G. Maier, A. Feldmann, S. Uhlig, Pitfalls in http traffic measurements and analysis, in: International Conference on Passive and Active Network Measurement, Springer, 2012, pp. 242–251.

[12] B. Thomas, R. Jurdak, I. Atkinson, Spdying up the web, Communications of the ACM 55 (12) (2012) 64–73.

[13] L. Shuai, G. Xie, J. Yang, Characterization of http behavior on access networks in web 2.0, in: Telecommunications, 2008. ICT 2008. International Conference on, IEEE, 2008, pp. 1–6.

[14] J. Sherry, C. Lan, R. A. Popa, S. Ratnasamy, Blindbox: Deep packet inspection over encrypted traffic, in: ACM SIGCOMM Computer Communication Review, Vol. 45, ACM, 2015, pp. 213–226.

[15] X. Yuan, X. Wang, J. Lin, C. Wang, Privacy-preserving deep packet inspection in outsourced middleboxes, in: Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on, IEEE, 2016, pp. 1–9.

[16] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, P. Steenkiste, Multi-context tls (mctls): Enabling secure in-network functionality in tls, in: ACM SIGCOMM Computer Communication Review, Vol. 45, ACM, 2015, pp. 199–212.

[17] S. Loreto, J. Mattsson, R. Skog, H. Spaak, G. Bourg, D. Druta, M. Hafeez, Explicit trusted proxy in http/2.0 (2014).
URL `https://tools.ietf.org/html/draft-loreto-httpbis-trusted-proxy20-01`

[18] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, A. Nucci, Per-user policy enforcement on mobile apps through network functions virtualization, in: Proceedings of the 9th ACM Workshop on Mobility in the Evolving Internet Architecture, MobiArch '14, ACM, New York, NY, USA, 2014, pp. 37–42. doi:10.1145/2645892.2645896.
URL `http://doi.acm.org/10.1145/2645892.2645896`

[19] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, P. Steenkiste, The cost of the s in https, in: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, ACM, 2014, pp. 133–140.

[20] OpenWrt: Linux distribution for embedded devices.
URL `https://openwrt.org`

[21] ApacheBench.
URL http://httpd.apache.org/docs/2.2/programs/ab.html

[22] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, RFC 2018 - TCP Selective Acknowledgment Options (1996).
URL https://www.ietf.org/rfc/rfc2018.txt

[23] M. Allman, V. Paxson, E. Blanton, TCP Congestion Control (2009).
URL https://www.ietf.org/rfc/rfc5681.txt

[24] M. Mellia, R. L. Cigno, F. Neri, Measuring ip and tcp behavior on edge nodes with tstat, Computer Networks 47 (1) (2005) 1–21.

[25] Tinyproxy.
URL http://tinyproxy.github.io

[26] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1 (1999).
URL https://tools.ietf.org/html/rfc2616.txt

[27] K9 Web Protection.
URL http://www1.k9webprotection.com

[28] Cisco Umbrella.
URL https://umbrella.cisco.com/use-cases/web-filtering

[29] Blue Coat WebFilter.
URL https://www.bluecoat.com/products/webfilter

[30] L.-D. Chou, Z. He, D. C. Li, H.-F. Chen, J.-J. Su, C.-Y. Chen, H.-C. Wei, C.-J. Li, Design and implementation of content-based filter system on embedded linux home gateway, in: Advanced Communication Technology (ICACT), 2012 14th International Conference on, IEEE, 2012, pp. 1046–1051.

[31] N. Herbaut, D. Negru, G. Xilouris, Y. Chen, Migrating to a nfv-based home gateway: introducing a surrogate vnf approach, in: Network of the Future (NOF), 2015 6th International Conference on the, IEEE, 2015, pp. 1–7.

[32] R. Cziva, S. Jouet, D. P. Pezaros, Roaming edge vnfs using glasgow network functions, in: Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, ACM, 2016, pp. 601–602.

[33] M. Baldi, R. Bonafiglia, F. Risso, A. Sapio, Modeling native software components as virtual network functions, in: Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, ACM, 2016, pp. 605–606.

[34] F. Sánchez, D. Brazewell, Tethered linux cpe for ip service delivery, in: Network Softwarization (NetSoft), 2015 1st IEEE Conference on, IEEE, 2015, pp. 1–9.

[35] P. B. Val, M. G. Valls, M. B. Cuado, A simple data-muling protocol, IEEE Transactions on Industrial Informatics 10 (2) (2014) 895–902. doi:10.1109/TII.2013.2290893.

[36] P. Basanta-Val, M. Garca-Valls, A distributed real-time java-centric architecture for industrial systems, IEEE Transactions on Industrial Informatics 10 (1) (2014) 27–34. doi:10.1109/TII.2013.2246172.

[37] z. lv, H. Song, P. Basanta-Val, A. Steed, M. Jo, Next-generation big data analytics: State of the art, challenges, and future research topics, IEEE Transactions on Industrial Informatics PP (99) (2017) 1–1. doi:10.1109/TII.2017.2650204.

[38] P. Basanta-Val, N. C. Audsley, A. J. Wellings, I. Gray, N. Fernndez-Garca, Architecting time-critical big-data systems, IEEE Transactions on Big Data 2 (4) (2016) 310–324. doi:10.1109/TBDATA.2016.2622719.