Hindawi
Scientific Programming
Volume 2019, Article ID 9067512, 11 pages
https://doi.org/10.1155/2019/9067512

Research Article

A Practical Approach to Protect IoT Devices against Attacks and Compile Security Incident Datasets

Bruno Cruz,[1] **Silvana Gómez-Meire**,[1] **David Ruano-Ordás**,[1,2,3,4] **Helge Janicke**,[3,4] **Iryna Yevseyeva**,[3,4] **and Jose R. Méndez**[1,2]

[1]*Department of Computer Science, University of Vigo, ESEI—Escuela Superior de Ingeniería Informática, Edificio Politécnico, Campus Universitario As Lagoas s/n, 32004 Ourense, Spain*
[2]*SING Research Group, Galicia Sur Health Research Institute (IIS Galicia Sur), SERGAS-UVIGO, Vigo, Spain*
[3]*Cyber Technology Institute, School of Computer Science and Informatics, De Montfort University, Gateway House 5.33, The Gateway, LE1 9BH Leicester, UK*
[4]*Faculty of Computing, Engineering & Media (CEM), De Montfort University, Leicester, UK*

Correspondence should be addressed to Jose R. Méndez; moncho.mendez@uvigo.es

Received 27 April 2019; Revised 26 June 2019; Accepted 7 July 2019; Published 29 July 2019

Guest Editor: Daniele D'Agostino

The Internet of Things (IoT) introduced the opportunity of remotely manipulating home appliances (such as heating systems, ovens, blinds, etc.) using computers and mobile devices. This idea fascinated people and originated a boom of IoT devices together with an increasing demand that was difficult to support. Many manufacturers quickly created hundreds of devices implementing functionalities but neglected some critical issues pertaining to device security. This oversight gave rise to the current situation where thousands of devices remain unpatched having many security issues that manufacturers cannot address after the devices have been produced and deployed. This article presents our novel research protecting IOT devices using Berkeley Packet Filters (BPFs) and evaluates our findings with the aid of our Filter.tlk tool, which is able to facilitate the development of BPF expressions that can be executed by GNU/Linux systems with a low impact on network packet throughput.

1. Introduction and Motivation

The evolution of Internet and communication networks from their emergence in the sixties to today has enabled a revolution in the way people and businesses interact. People today communicate worldwide using mobile devices, which have a reliable broadband (4G) Internet connection. Despite these great advances, Aceto et al. [1] note that network outages are still a challenge to solve because they are frequent, hard to fix, expensive, and, in particular, poorly understood by users. Whilst there exists a variety of problems surrounding network availability (Aceto et al. [1]), this study presents a proposal to avoid or at least minimize the effects of problems caused by software attacks through networks, including worms [2, 3] and remote attacks to exploit server vulnerabilities [4].

Patching avoids the compromise of target systems through, e.g., malware and vulnerability exploits. However, the growth of Internet of Things (IoT) applications running on devices, that frequently do not support patching, using Internet and TCP/IP networks for communication purposes, limits this possibility. Moreover, software upgrades are not always immediately available when the vulnerability is discovered, as patch development and distribution depend on developer circumstances. The existence of proactive defense mechanisms [5] capable of mitigating risks associated with unpatched components within an otherwise trusted TCP/IP network would be very valuable. In this study, we take advantage of the firewall support of operating systems to develop a highly efficient mechanism to detect and bring together information about malicious network traffic.

Firewalling support has become an essential feature of modern operating systems. The use of firewalls is one of the easiest mechanisms to manage network defense. However, its effectiveness is clearly limited to protect IoT devices against malware and vulnerability exploiting [6]. In the well-known Linux operating system, firewall capabilities have been provided primarily through packet filtering technology and have evolved from a netfilter ipfw system port (included in Linux kernel 1.1) to netfilter/iptables (included in Linux 2.4 kernel series). This evolution entailed the introduction of significant innovations such as the tracking of TCP connections or the possibility of altering packets in transit (mangle table). Despite the popularity of these filters, netfilter/iptables firewalling subsystem will be replaced in order to speed up the filtering process and increase the information achieved for each packet to filter (such as payload information).

Wireshark capture filters [7] are defined by using libpcap filter language. Filter examples that are designed to detect some worms and exploits are available in Wireshark Wiki [8] showing the power of this filter syntax. The syntax of capture filters is commonly known as Berkeley Packet Filter (BPF) and is supported in the kernel of most UNIX-like operating systems. This syntax is also implemented by libpcap/Winpcap to be used at the user level in tools such as Wireshark. BPF [9] was first introduced in 1990 as a tool for capturing and filtering network packets that matched specific rules. BPF support was included in Linux kernel by implementing a small virtual machine that runs compiled BPF programs injected from user-space [10]. Later, a BPF Just-In-Time (JIT) compiler was added to speed up the performance of the execution of bytecodes. Currently, BPF can be loaded for its execution into kernel with different tools to execute different tasks, such as system monitoring (trough using perf tool), network traffic control and quality of service (through tc tool), and packet filtering (through ip link tool included in iproute2 suite or iptables).

Due to its flexibility, BPF has been used by important technology companies such as Google, Facebook, Cloudflare, and Netflix to address network security issues, load-balancing, traffic-filtering, and monitoring [11–14]. A comparison of the filtering performance achieved by a BPF-based filter (BPFilter), iptables, and nftables has also been provided in other studies [11, 15] showing that BPFilter runs up to 5 times faster than iptables. This scenario led to the consideration of BPF as a reliable candidate to replace iptables (and nftables) as the kernel firewall subsystem for Linux [11]. However, despite the fact that BPF syntax is more powerful than that offered by current Linux firewalls, BPFilter only takes advantage of the BPF virtual machine to speed up rules created by older tools. Majkowski [16, 17] demonstrated how to take advantage of BPF in conjunction with iptables to filter packages and define new chains. These works allowed system administrators to take advantage of the rich syntax and efficient execution of BPF expressions to filter packages in real environments and protect IoT devices against malware and vulnerability exploiting.

We developed Filter.tk to work in conjunction with these tools. Filter.tk is a framework to complete the full lifecycle (creation, debugging, and testing) of BPF iptables-compliant pattern design for mitigating both worm and exploit attacks. The development of patterns will be useful for the future creation of a BPF rules database usable in the form of well-known community collaboration products such as Ansible Galaxy [18, 19] or DockerHub [20], where users can share BPF to protect IoT devices, computers, and software against worm and exploit attacks. Additionally, the information about harmful network packets can be uploaded to centralized repository for research purposes. Particularly, this data, if compiled worldwide, could allow the identification of security threats and help in the identification of new offensive packet patterns.

The remainder of this paper is structured as follows: Section 2 introduces the state of the art in well-known worms, security vulnerabilities, and IoT security. Section 3 introduces our proposal to address both the protection of devices against security vulnerabilities (and hence, worm attacks exploiting those vulnerabilities), and the compilation of security incident datasets in the context of IoT while Section 4 is centered on discovering the utility of the toolset through case studies. Finally, Section 5 presents the main conclusions of the work and outlines the directions for future research.

2. State of Art

During the early 2000s, Internet worms became very popular due to the effects of well-known worms such as Code Red (versions 1 and 2), Nimda, SQL Slammer, or Blaster. Some of these worms are compiled in the work of Qing and Wen [2]. However, due to the increased awareness of users and developers about the importance of security, this kind of malicious software is solely spread in P2P networks and operates in a passive form [21]. Instead of performing an active search to infect computers, passive worms require human intervention, i.e., by downloading an infected file from a P2P network, to replicate themselves. Despite the propagation of passive worms in P2P networks mainly connected with the illegal downloading of software and multimedia materials, the dissemination of these Internet worms and their mitigation has been fairly well discussed in previous studies [21–29]. The detection of new vulnerabilities allowing remote exploitation is a very active area as evidenced by the latest exploits published in exploit-db [4]. Although the existence of vulnerabilities allowing the execution of remote commands could provide a mechanism for the dissemination of worms, the quick response of software development teams to provide security patches discourages malware developers from designing new worms. With this in mind, the goal of this work is to mitigate attacks exploiting software vulnerabilities, with a special interest in those targeting an IoT device.

Many IoT applications and devices have become available for smart home automation. Querying “remote” “hardware” exploits in exploit-db and other similar databases resulted in a number of exploitable vulnerabilities in

well-known products (such as intelligent TVs, cameras, etc.). This shows that IoT developers have been prioritizing the development and creation of functionalities for most demanding users while frequently neglecting security considerations.

A few works have addressed issues in IoT security, such as the use of block-chain communications [30–33]. These usually refer to security issues pertaining to confidentiality, integrity, and availability in the communications between IoT devices and IoT. The work of Ammar et al. [34] provides a critical review of eight well-known IoT frameworks with special emphasis on security issues (analyzing models and approaches provided for ensuring security and privacy, pros and cons of each framework in terms of fulfilling the security requirements and meeting the standard guidelines, and identifying design flaws). Wood and Stankovic [35, 36] provide studies about network issues. Particularly, the former work is centered in security-related issues about IoT communication protocols whilst the later analyzes denial of service threats in IoT environments.

Wack et al. [37] review the risk of platform software/firmware vulnerabilities that enable the reception of malicious attacks. To the best of our knowledge, there is no research work focused on the prevention, management, and response to vulnerability exploiting and worm attacks in IoT. Closing this gap, we studied how to take advantage of firewalling schemes to implement these protections.

2.1. OS Firewalling Support. Common OS firewalls, such as those that can be implemented through GNU/Linux kernel firewalling subsystem, are usually implemented as packet filters [37, 38], which consist of a default policy for packets and a sequence of rules that define the actions performed on packets when they satisfy certain conditions. Specifically, each firewalling rule contains a triggering condition, usually a simple condition or the logical AND of simple conditions, together with an action to execute when the rule is triggered. Triggering conditions are defined over the second, third, and fourth TCP/IP layers. The support for stateful inspection of connections is available for kernel versions 2.4 and above [39].

The first GNU/Linux firewall generation was included on 1.1 kernel through an implementation of ipfw functionalities contributed by Alan Cox [40]. The ipfwadm user-space tool was used to configure the ipfw services offered by the kernel [41]. These kernels allowed defining three different firewall filters to handle (i) input packets (*-I* ipfwadm argument), (ii) output packets (*-O*), and (iii) forwarded packets (*-F*, used in conjunction with ip_forwarding feature). Accept, deny (discard the packet), and reject actions were used either for rules (-a command parameter) or as default policy (-p). In order to create and design the trigger condition of each rule, the system administrator can test for the protocol (TCP, UDP, ICMP, or IP), the port (for TCP and UDP) or the ICMP type. Logging is supported through the *-o* modifier.

The support for ipfw was replaced by ipchains in the 2.2 version of the kernel [42]. One of the most important changes in the ipchains scheme was the introduction of chains to help reduce the computational cost and facilitate its design [43]. As opposed to the others, ipchains firewalls include INPUT, OUTPUT, and FORWARD chains, which bring together filtering rules applied to packets where the current computer is the destination, the origin, or a router for the packet, respectively. Each firewall chain is composed of a ruleset and a default policy. The default policy is applied to packets that do not match any rule. The existence of default policies allows defining firewalls using two different schemes: (i) accept all except those packets explicitly denied or (ii) deny all except those packets explicitly accepted. Of these, the latter is advisable for security reasons.

New functionalities offered by ipchains with regard to ipfw were quite limited, so it was quickly replaced by iptables (in Linux 2.4 series) [44]. Iptables/netfilter included the table concept to bring together chains with similarities. Iptables included the firewall tables filter, nat, and mangle. The first one included three chains to support the filtering of input, output, and forwarded connections (INPUT, OUTPUT, and FORWARD respectively). The NAT table is composed of PREROUTING and POSTROUTING chains to add rules to change destination or source addresses, respectively. To this end, rules included in these chains could only use DNAT, SNAT, REDIRECT, or MASQUERADE actions. Finally, MANGLE table allows marking packages for further processing and modifying some parameters of packets including TOS or TTL. These actions could be executed by using MARK, TOS, and TTL actions. Finally, iptables brought the stateful packet inspection to Linux firewalls making it possible to determine whether a packet belongs to an established TCP connection (-m state--state = ESTABLISHED) or, conversely, is connected with other previous packets (-m state--state = RELATED).

Iptables have been widely used to implement packet filters on Linux for many years [45]. However, some limitations of iptables, such as the existence of a unique action for a rule or the complexity of the syntax, led to the creation of other filtering frameworks. Hence, nftables emerged as an iptables replacement on kernel version 3.13 (2013) [44]. Nftables included a completely new and fresh syntax that avoided the need to use hyphens and the uppercase/lowercase flags. The use of nftables allows tables and chains to be created with specific names and associated with hooks, thus avoiding the strict tables/chains structure defined by iptables.

Despite the new functionalities of nftables, most Linux users continue using the old iptables framework, in part due to the numerous changes in the syntax which hindered adoption. Some translation utilities were introduced to aid in the migration from iptables to nftables [46]. In addition, nftables work as a sequential filter whereby every packet is matched one by one against a list of rules. The speed of checking the rules is quite limited (up to three times slower than using BPF) [11], which led to the emergence of bpfilter as a new Linux firewalling subsystem [47] able to outperform the speed of previous

filtering alternatives [11]. Bpfilter has been added experimentally to Linux kernel 3.18, now allowing nftables and iptables rules to be executed by Linux kernel as BPF.

Standard definitions within BPF only allow current packet filtering firewall [48] schemes to analyze some information from packet headers (such as IP and MAC addresses, ports, TCP flags, ICMP types, etc.) and packet state. Additional new features would improve the firewalling performance, such as analyzing the payload of the packet or the information about application layer protocols. These features are frequently included in deep packet inspection techniques [49, 50] but are often too slow to be included in standard firewalls. In order to provide a deep description of packets on the firewall layer and quickly evaluate them, the use of BPF language together with the BPF virtual machine subsystem included in the current versions of Linux kernel seems to be an elegant solution, especially as BPF had been used before to accomplish similar difficult network tasks with low computational effort [11].

Despite its performance and low computational costs, developing a BPF-based firewall able to exploit full packet data (headers and payload) is a hard task that would require both the existence of tools to aid in the development of conditions and packet datasets. Caploader [51] and Wireshark/tcpdump [7], which can also be integrated with NDPI [52], are capable of loading and analyzing packets included in PCAP files and check whether a BPF expression match them. These tools can be successfully executed with large collections of packets, such as that shared by Netresec [53]. However, the design of BPF filters is not easy and should be simplified to impact on real-world firewall applications. Similarly, the evaluation of BPF filters should be automated to improve performance. Both the simplification and automation have been addressed by our Filter.tlk toolset and are the main contribution of this work. Filter.lk's functionality and its practical use are described in the next section.

3. Filter.tlk

This section provides a comprehensive description of the design architecture of Filter.tlk [54] tool and documents the process of creating customized filters to classify network traffic according to the content of the packets.

Filter.tlk comprises three different utilities to aid in the creation of BPF filters: (i) an interface to design BPF filtering conditions, (ii) a Wireshark LUA plugin to automate the testing of BPF filters with PCAP packet datasets, and (iii) a script to easily compile BPF filters and create iptables rules. Figure 1 shows the different components included in Filter.tlk and their use in a real environment.

As we can deduct from Figure 1, the design of a firewall rule with Filter.tlk comprises three stages that are made with different tools included in the package. We begin by taking advantage of the BDAT (BPF design aid tool) to design a filtering condition to detect a certain kind of packet. The designed BPF filters can then be tested with different packet sets (a set of packets matching the pcap filter and others mismatching the pcap filter) using BPF Testing tool (BTT). BTT is able to easily assess the quality of an input filter by using different datasets. Once BPF rules have been tested, they can be easily transformed into iptables rules using IPTables Rule Builder (IPTRB) script.

BDAT is responsible for creating BPF filters as conditions defined from transport and network layers (see Figure 1). By using BDAT through a simple graphical interface, we can create a Boolean BPF filter evaluating expressions related to network or transport headers and payloads (UDP, TCP, IP, ICMP). In order to create header conditions, users must select the field of the header on which they want to establish the condition that the filter must fulfill. Once the condition has been defined, the user can continue adding new conditions for the same filter or create a new one. Once all filters are defined, they can be exported to a file for testing in BTT (BPF Testing Tool). As an example, Figure 2 shows how administrators can easily incorporate a condition about a HTTP POST request by specifying conditions about TCP payload.

As depicted in Figure 2(a), we selected the first four octets from TCP payload for comparison purposes. BDAT allows selecting one, two, or four octets from each word (32 bits) to check the condition. The next step of the wizard (see Figure 2(b)) allows to easily define the value using Hexadecimal, ASCII, or Decimal notations. In order to compare any octet from payload (and options/padding), the offset value (highlighted in red in Figure 2(a)) can be edited to the desired value. Please note that the designed condition is provided as example and should be complemented with a "header length" value of five to ensure the absence of options/padding field. In the next step of the wizard, the current BPF condition rule is added to the whole BPF filter, allowing the generation of filters comprising multiple tests.

BTT is a plugin for Wireshark that applies a filter or set of filters over a pcap file. As a result, we obtain information for each applied filter about the number of packages analyzed, accepted, and rejected. A set of pcaps with the packets accepted by the filter grouped by the destination IP is also provided for debugging purposes. In order to detect errors, each rule should be tested using a pcap database containing only the packets that should be captured (ensuring a result of 0 rejected is achieved) and another one containing normal network traffic (guaranteeing a result of 0 accepted is achieved).

Finally, IPTRB (IPTables Rule Builder) script can transform the BPF uncompiled filters into full featured IPTables rules. The process is guided by an intuitive libncurses-based graphical user interface that allows customizing the generated rule. The rule can be generated for filtering and/or harmful packet logging purposes. A scheduled task (i.e., crontab) could be periodically executed (for instance once a day) to upload the compiled information (logs) to a centralized repository for its further analysis.

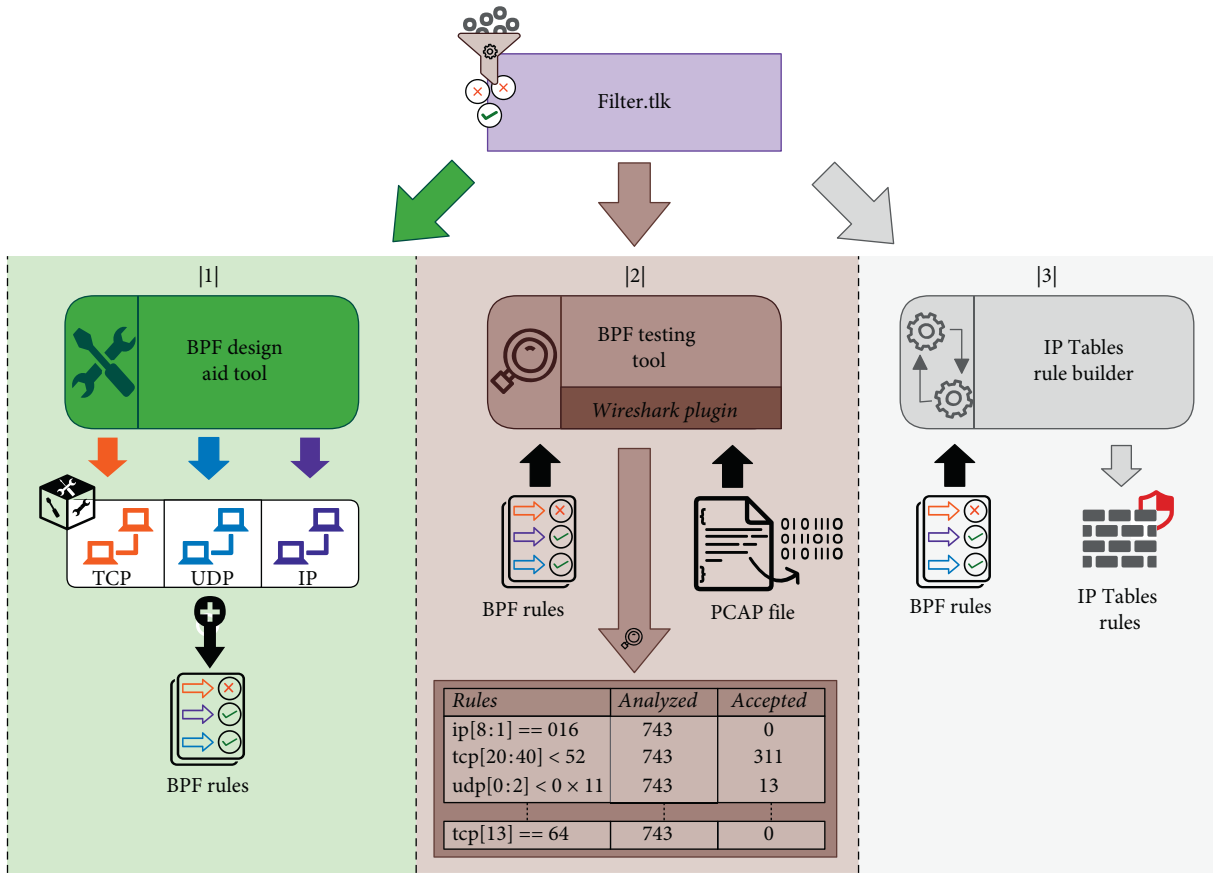


Figure 1: Filter.tlk architecture.

3.1. Filter.tlk Implementation. This subsection provides a brief description of the most relevant implementation issues for the development of each tool included in Filter.tlk.

BDAT was designed as a Java standalone application, which can easily be executed using any Java Virtual Machine implementation. The interface was designed using JFC/Swing library [55].

BTT is a Wireshark plugin that was implemented using the Lua programming language [56], which is supported by Wireshark for the development of new functionalities, such as the creation of dissectors or listeners [57]. The dissectors are intended to analyze part of the data of a packet, while the listeners are used to count the number of occurrences of an event; for example, the number of packets matching a filter. In this study, we used Lua language to implement a Wireshark listener to evaluate filters and count packets fitting the target BPF condition (accepted) or not (rejected).

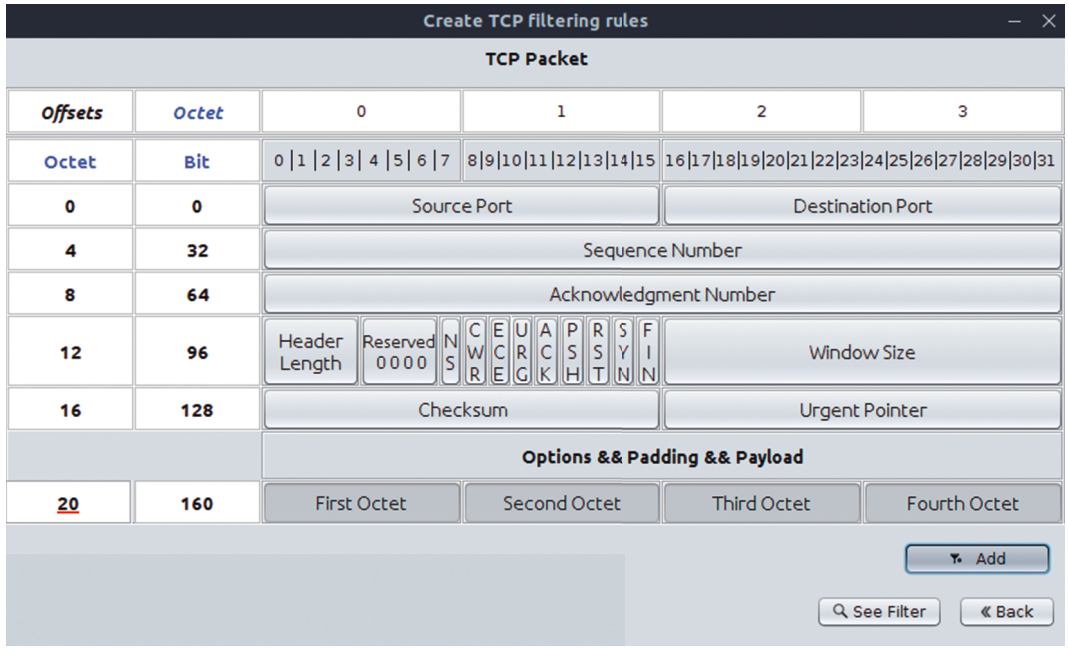
IPTRB is a bash script that combines the use of dialog command [58] to provide an easy-to-use intuitive graphical user interface. Moreover, the compilation of BPF rules into bytecode is done by using tcpdump [59] functionalities.

Finally, we used Ansible [19] (a well-known IT Automation tool) to automate the installation of Filter.tlk in all supported platforms. Ansible is a popular IT automation tool whose main features are (i) avoiding the need of scripts and/or custom code to deploy and update applications and (ii) replacing agents on remote systems by standard SSH tools. The installation script was provided for Debian-based GNU Linux distributions.

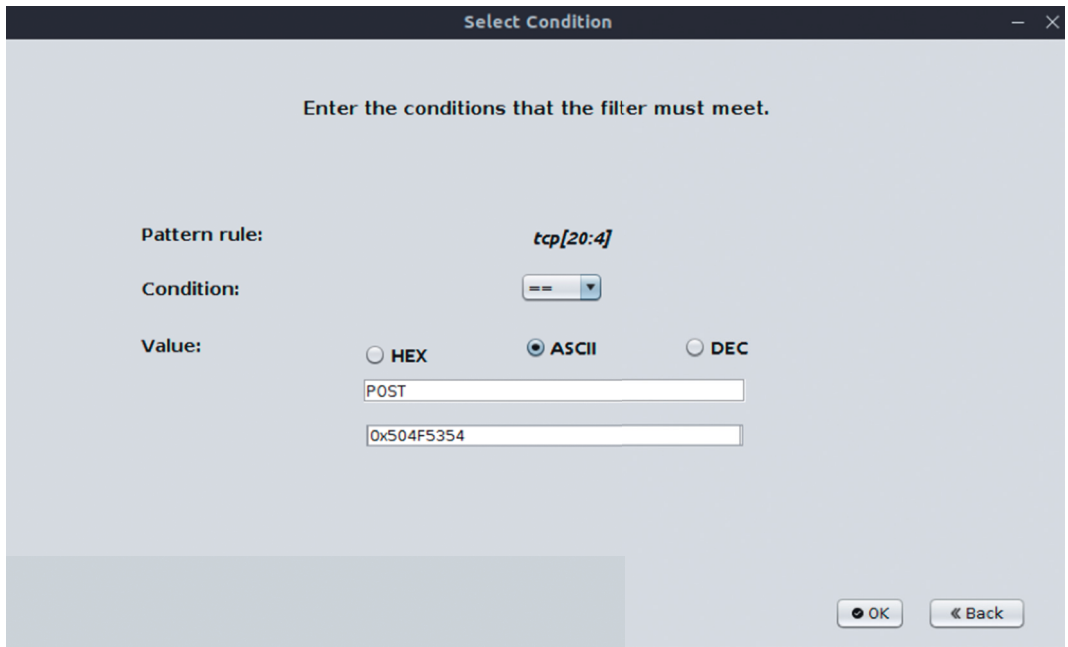
3.2. Deployment of Generated Filters. To take advantage of expressions (iptables rules and BPF) generated using our BPF framework, we consider two different scenarios: (i) IoT devices using a GNU Linux-based software/firmware (ii) and other IoT devices with no BPF/iptables support. In the first scenario, iptables rules can be directly integrated into the firmware to protect them against malicious attacks. We are working on the development of a service to share BPFs together with a tool able to automatically download and upgrade BPFs for different IoT devices.

Although GNU/Linux is present in some devices, there are many appliances running other operating systems where the execution of BPF is not possible. Taking this into account, we are working on the design of a small bridge router (brouter) [60] device running GNU/Linux and ebtables. A brouter is a device that is able to transparently forward all traffic between two ethernet interfaces and allows the inclusion of filtering rules for network interfaces. This solution would be applicable for IoT devices connected to the network through an ethernet connection.

The main weakness of using BPF filters to protect devices against attacks is that we are unable to protect 802.11-based (WLAN, wireless local area network) IoT devices that do not run GNU/Linux.



(a)



(b)

Figure 2: BPF rule definition process. (a) Select TCP payload octets. (b) Establish values for octets.

Next section presents a comprehensive practical example describing the process of using the Filter.tlk tool to design a filter capable of detecting and filtering two important vulnerabilities recently discovered on well-known IoT devices.

4. Experiments

In this section, we test the filter designed to detect and filter attacks using two vulnerabilities in two well-known IoT devices that allow the remote execution of arbitrary commands: (i) LG Supersign TVs, and (ii) ASUS ADSL Router DSL-N12E_C1. The next subsection shows the work environment prepared in order to generate high-quality BPF patterns. Moreover, subsection 4.2 presents the experimental protocol and results of our case studies while subsection 4.3 measures the impact of the use of these filters in the performance of IoT devices. Finally, subsection 4.4 shows how to compile and take advantage of the information gathered by IoT devices using Filter.tlk for scientific purposes.

4.1. Configuring the Working Environment. In order to generate high-quality BPF expressions that describe the pattern of vulnerability exploitation, the use of a large packet database for testing purposes is advisable. Fortunately, many publicly available packet datasets can be freely downloaded from the Internet. Table 1 compiles a list of useful datasets.

From the datasets included in Table 1 and other sources, we built up a group of packets that would be used to ensure that inoffensive network requests are not captured by the designed BPF expression.

We decided to study and generate BPF filters for two vulnerabilities of well-known IoT devices. In order to determine the quality of the BPF expressions created using a BTT Lua script, we used as negative samples the conjunction of all packets from sources introduced in Table 1 and other legitimate packet sets compiled by us. In order to aggregate all negative samples in a single pcap file, we combined all sources using the mergecap [64] tool provided by Wireshark.

Table 1: Publicly available datasets.

Dataset	Number of packets	Number of files	Format	Short description
Contagiodump [61]	988898	1154	pcap zipped	Collect malicious and exploit pcaps from various public resources (2013–2015)
Malware traffic analysis [62]	2445211	1291	pcap zipped	Malicious network traffic (2013–2018)
GTISK PANDA malrec [63]	100201	373	pcap	Malware samples run in PANDA (2018)

4.2. Executing the Experiment. Recently, a vulnerability allowing remote execution of arbitrary commands appeared on LG SuperSign TVs (CVE-2018-17173) [65, 66]. These smart TVs include a CMS running on the top of LG webOS 3.3 (a Linux-based OS). The discovered vulnerability allows remote code execution (by achieving a reverse shell connection) by taking advantage of the URL used to see thumbnails of the user images. We used the exploit versions to generate a pcap file capturing the attacks. The Filter.tk comprises a three-stage operation. The first step designs the filtering condition to detect the packets, and the second step tests the BPF filter with a set of packets captured in a pcap filter and with a set of normal packets. The third step converts the BPF into iptables rules. The generated BPF expression is shown in Table 2. These BPF expressions could be directly included in LG webOS to protect the TV.

The second analysis is about a remote code execution vulnerability in ASUS DSL-N12E_C1 router, specifically in firmware version 1.1.2.3_345 (CVE-2018-15887) [67]. This vulnerability has been classified as critical because it allows the execution of arbitrary code using an unknown function of the file "Main_Analysis_Content.asp." A remote attacker can then access the router as a privileged user via telnet application and run OS commands. Again, we take advantage of our framework to generate BPF expressions to filter this type of attack. The generated BPF is shown in Table 3.

As shown in Tables 2 and 3, an iptables command can be easily generated from a BPF expression to drop and log packets that match it. Although iptables can only check expressions in the header of network packets, BPF expressions make it possible to examine information included in both packet headers and payload in order to find any potential exploitation of vulnerabilities. The second generated iptables rule allows for storing security information that can be uploaded to a centralized repository for its further analysis. In next section, we evaluate the performance impact on the IoT devices when using these filters.

4.3. Impact on Filter Throughput. We assessed the impact of using BPF filters on IoT devices in order to determine if they could be successfully used to protect IoT devices against network vulnerability exploitation. To perform this analysis, we used an Apache web server installed on a Raspberry Pi 2 Model B [68].

We leveraged the functionalities of Apache HTTP server benchmarking [69] and GNU parallel [70] tools to evaluate the impact of using BPF firewalls in IoT hardware. Using these tools, we benchmarked the execution of two parallel tests making 10000 HTTP requests distributed in 10 threads, with 1000 requests per thread. The average of measurements made for parallelized tests is provided as result. For comparison purposes, we used the generated BPF expressions for the two case studies shown before. Table 4 compares the performance between the absence of attack protections and the usage of two BPF filtering rules.

The results compiled in Table 4 show that the performance impact when using BPF filters is quite limited and will not severely affect the overall operation of IoT devices. We analyzed the impact of progressively adding BPF rules to the filter by adding up to 100 new rules and measured the transfer rate after each BPF expression was added (see Figure 3). As long as the performance is highly influenced by the presence of additional traffic in network and other processes consuming CPU, we plotted a trend line to observe the degradation.

As can be seen from Figure 3, the throughput degradation is close to zero when using up to 50 (nonfitting) BPF rules. However, the inclusion of more than 50 rules clearly damages the performance of GNU/Linux firewalling system and would require the usage of additional iptables speedup strategies, such as the creation of additional chains [71] and counters-based optimizations [72].

One of the most interesting features of Filter.tlk framework is the compilation of information about worldwide IoT security incidents. The information gathered could be successfully analyzed using machine learning

Table 2: BPF expression to mitigate CVE-2018-17173 vulnerability.

BPF expression	ip[2 : 2] > 0x008A and ip[9] == 0x06 and tcp[2 : 2] == 0x2378 and tcp[32] == 0x47 and tcp[77 : 4] == 0x3d253237 and tcp[81 : 4] == 0x2532302d and tcp[85] == 0x3b			
	BPF assembler code			Bytecode
(000) ldh	[12]			22
(001) jeq	#0x800	jt 2	jf 21	40 0 0 12
(002) ldh	[16]			21 0 19 2048
(003) jgt	#0x8a	jt 4	jf 21	40 0 0 16
(004) ldb	[23]			37 0 17 138
(005) jeq	#0x6	jt 6	jf 21	48 0 0 23
(006) jeq	#0x6	jt 7	jf 21	21 0 15 6
(007) ldh	[20]			21 0 14 6
(008) jset	#0x1fff	jt 21	jf 9	40 0 0 20
(009) ldxb	4 ∗ ([14]&0xf)			69 12 0 8191
(010) ldh	[x + 16]			177 0 0 14
(011) jeq	#0x2378	jt 12	jf 21	72 0 0 16
(012) ldb	[x + 46]			21 0 9 9080
(013) jeq	#0x47	jt 14	jf 21	80 0 0 46
(014) ld	[x + 91]			21 0 7 71
(015) jeq	#0x3d253237	jt 16	jf 21	64 0 0 91
(016) ld	[x + 95]			21 0 5 1025847863
(017) jeq	#0x2532302d	jt 18	jf 21	64 0 0 95
(018) ldb	[x + 99]			21 0 3 624046125
(019) jeq	#0x3b	jt 20	jf 21	80 0 0 99
(020) ret	#262144			21 0 1 59
(021) ret	#0			6 0 0 262144
				6 0 0 0

Iptables commands

iptables -t filter -A INPUT -m bpf --bytecode "22,40 0 0 12,21 0 19 2048,40 0 0 16,37 0 17 138,48 0 0 23,21 0 15 6,21 0 14 6,40 0 0 20,69 12 0 8191,177 0 0 14,72 0 0 16,21 0 9 9080,80 0 0 46,21 0 7 71,64 0 0 91,21 0 5 1025847863,64 0 0 95,21 0 3 624046125,80 0 0 99,21 0 1 59,6 0 0 262144,6 0 0 0" -j DROP

iptables -t filter -A INPUT -m bpf --bytecode "22,40 0 0 12,21 0 19 2048,40 0 0 16,37 0 17 138,48 0 0 23,21 0 15 6,21 0 14 6,40 0 0 20,69 12 0 8191,177 0 0 14,72 0 0 16,21 0 9 9080,80 0 0 46,21 0 7 71,64 0 0 91,21 0 5 1025847863,64 0 0 95,21 0 3 624046125,80 0 0 99,21 0 1 59,6 0 0 262144,6 0 0 0" -j LOG --log-prefix "Filter.tlk"

techniques to provide worthwhile knowledge about (i) the origin of the threat, (ii) better patterns for traffic-filtering, or (iii) the threat scale. Studying information about systems from which the attack is performed, we can successfully identify worms exploiting a certain vulnerability, the presence of an individual hacker, and the execution of Distributed Denial of Service attacks or botnets.

Offering IoT users a product to protect their devices against attacks, whilst at the same time achieving information about dangerous offensive network packets targeting IoT products, will replicate a threat response model undertaken by traditional antivirus products. This knowledge allows the identification of better and perhaps simpler BPF patterns that can be used for network intrusion detection.

5. Conclusions and Future Work

In this paper, we have introduced an easy-to-use framework designed to aid in the development of fast firewalls based on using BPF, which can be executed by using standard firewall capabilities included in the Linux kernel (IPTables/netfilter). These firewalls have been specifically conceived to protect IoT devices against the exploitation of remote vulnerabilities. Since the use of BPF bytecode can drastically speed up the execution of firewalls, we designed a collection of tools to facilitate the inclusion of BPF into firewalling rules. An experiment was carried out for the application of the introduced toolset.

Since BPF is one of the most efficient forms of filtering traffic, it provides a reliable solution for filtering in the context of IoT. Although, the use of specific BPF filters allows using payload information included in packets, it can only be directly implemented in devices using a GNU/Linux-based firmware. We are currently working on the design of specific hardware to overcome the limitations of nonGNU/Linux ethernet IoT devices and on the development of a package manager to automatically download BPF filtering strings and configure the firewall.

One of the most relevant functionalities of this scheme is the ability to easily build a dataset with the security incidents occurred in worldwide IoT devices, such as VizSec [73]. Future work will include mechanisms for analyzing them to achieve valuable security knowledge. We consider evolutionary computation as a candidate method for automatic filter generation through packet captures and consider DPI (deep packet inspection [52]) to be a reliable way of simplifying filtering conditions, since it allows access to application layer information to define matching

Table 3: BPF expression to mitigate CVE-2018-15887 vulnerability.

BPF expression	ip[2:2] > 0x0174 and ip[9] == 0x06 and tcp[2:2] == 0x0050 and tcp[32] == 0x47 and tcp[326:4] == 0x3d253630			
	BPF assembler code			Bytecode
(000) ldh	[12]			18
(001) jeq	#0x800	jt 2	jf 17	40 0 0 12
(002) ldh	[16]			21 0 15 2048
(003) jgt	#0x174	jt 4	jf 17	40 0 0 16
(004) ldb	[23]			37 0 13 372
(005) jeq	#0x6	jt 6	jf 17	48 0 0 23
(006) jeq	#0x6	jt 7	jf 17	21 0 11 6
(007) ldh	[20]			21 0 10 6
(008) jset	#0x1fff	jt 17	jf 9	40 0 0 20
(009) ldxb	4 * ([14]&0xf)			69 8 0 8191
(010) ldh	[x + 16]			177 0 0 14
(011) jeq	#0x50	jt 12	jf 17	72 0 0 16
(012) ldb	[x + 46]			21 0 5 80
(013) jeq	#0x47	jt 14	jf 17	80 0 0 46
(014) ld	[x + 340]			21 0 3 71
(015) jeq	#0x3d253630	jt 16	jf 17	64 0 0 340
(016) ret	#262144			21 0 1 1025848880
(017) ret	#0			6 0 0 262144
				6 0 0 0

Iptables commands

iptables -t filter -A INPUT -m bpf --bytecode "18,40 0 0 12,21 0 15 2048,40 0 0 16,37 0 13 372,48 0 0 23,21 0 11 6,21 0 10 6,40 0 0 20,69 8 0 8191,177 0 0 14,72 0 0 16,21 0 5 80,80 0 0 46,21 0 3 71,64 0 0 340,21 0 1 1025848880,6 0 0 262144,6 0 0 0" -j DROP

iptables -t filter -A INPUT -m bpf --bytecode "18,40 0 0 12,21 0 15 2048,40 0 0 16,37 0 13 372,48 0 0 23,21 0 11 6,21 0 10 6,40 0 0 20,69 8 0 8191,177 0 0 14,72 0 0 16,21 0 5 80,80 0 0 46,21 0 3 71,64 0 0 340,21 0 1 1025848880,6 0 0 262144,6 0 0 0" -j LOG --log-prefix "Filter.tlk"

Table 4: Performance impact when using BPF filters.

	No protection	With BPF (2 rules)
HTML transferred (bytes)	107010000	107010000
Concurrent time per request (ms)	1.9345	1.937
Time per request (ms)	19.3465	19.367
Time taken for tests (seconds)	19.3465	19.367
Total transferred (bytes)	109750000	109750000
Transfer rate (Kbytes/sec)	5539.905	5534.1

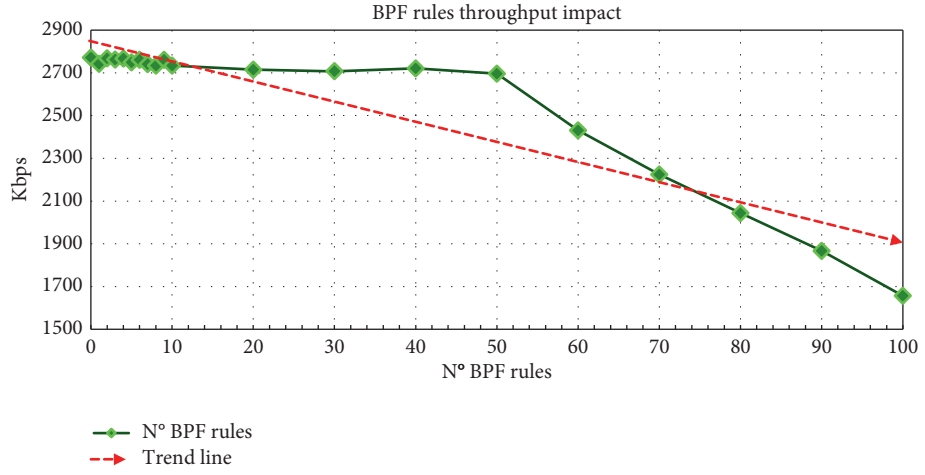


Figure 3: Analysis of the impact of BPF rules in throughput.

expressions. While DPI expressions cannot be directly included in BPF filters, we believe that they could be automatically transformed into simple BPF expressions to simplify the generation of BPF filters.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

D. Ruano-Ordás was supported by a postdoctoral fellowship from Xunta de Galicia (ED481B 2017/018). Additionally, this work was funded by the project Semantic Knowledge Integration for Content-Based Spam Filtering (TIN2017-84658-C2-1-R) from the Spanish Ministry of Economy, Industry and Competitiveness (SMEIC), State Research Agency (SRA), and the European Regional Development Fund (ERDF) and by the Consellería de Educación, Universidades e Formación Profesional (Xunta de Galicia) under the scope of the strategic funding of ED431C2018/55-GRC Competitive Reference Group. SING group thanks CITI (Centro de Investigación, Transferencia e Innovación) from University of Vigo for hosting its IT infrastructure.

References

[1] G. Aceto, A. Botta, P. Marchetta, V. Persico, and A. Pescapé, "A comprehensive survey on internet outages," *Journal of Network and Computer Applications*, vol. 113, pp. 36–63, 2018.

[2] S. Qing and W. Wen, "A survey and trends on Internet worms," *Computers & Security*, vol. 24, no. 4, pp. 334–346, 2005.

[3] D. E. Hiebeler, A. Audibert, E. Strubell, and I. J. Michaud, "An epidemiological model of internet worms with hierarchical dispersal and spatial clustering of hosts," *Journal of Theoretical Biology*, vol. 418, pp. 8–15, 2017.

[4] Offensive Security, Exploit Database, 2009.

[5] M. Ge, J. B. Hong, S. E. Yusuf, and D. S. Kim, "Proactive defense mechanisms for the software-defined Internet of things with non-patchable vulnerabilities," *Future Generation Computer Systems*, vol. 78, pp. 568–582, 2018.

[6] R. K. Deka, K. P. Kalita, D. K. Bhattacharya, and J. K. Kalita, "Network defense: approaches, methods and techniques," *Journal of Network and Computer Applications*, vol. 57, pp. 71–84, 2015.

[7] G. Combs, Wireshark Go Deep, 1998.

[8] Wireshark, Wireshark CaptureFilters, 2016.

[9] S. McCanne and V. Jacobson, "The BSD packet filter: a new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference*, p. 2, San Diego, CA, USA, January 1993.

[10] P. Anand, "An intro to using eBPF to filter packets in the Linux kernel," 2017, http://www.OpenSource.com.

[11] T. Graf, "Why is the kernel community replacing iptables with BPF?," November 2018, https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/.

[12] Ł. Makowski and P. Grosso, "Evaluation of virtualization and traffic filtering methods for container networks," *Future Generation Computer Systems*, vol. 93, pp. 345–357, 2019.

[13] G. Bertin, "XDP in practice: integrating XDP into our DDoS mitigation," in *Proceedings of the Technical Conference on Linux Networking*, p. 5, Seoul, South Korea, November 2017.

[14] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proceedings of the 1994 Winter USENIX Conference*, pp. 153–165, San Francisco, CA, USA, January 1994.

[15] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with Linux eBPF," in *Proceedings of the 2018 30th International Teletraffic Congress (ITC 30)*, pp. 209–217, Vienna, Austria, September 2018.

[16] M. Majkowski, BPF—the Forgotten Bytecode, 2014.

[17] M. Majkowski, Introducing BPF Tools, 2014.

[18] Read Hat, *Ansible Galaxy*, Read Hat, Inc., Raleigh, NC, USA, 2018.

[19] M. DeHaan, *Ansible is Simple IT Automation*, Read Hat, Inc., Raleigh, NC, USA, 2012.

[20] Docker Inc., *Docker Hub*, Docker, Inc., San Francisco, CA, USA, 2016.

[21] M. A. Rguibi and N. Moussa, "Hybrid trust model for worm mitigation in P2P networks," *Journal of Information Security and Applications*, vol. 43, pp. 21–36, 2018.

[22] F. Wang, Y. Zhang, and J. Ma, "Defending passive worms in unstructured P2P networks based on healthy file dissemination," *Computers & Security*, vol. 28, no. 7, pp. 628–636, 2009.

[23] T. Chen, X.-S. Zhang, H. Li, X.-D. Li, and Y. Wu, "Fast quarantining of proactive worms in unstructured P2P networks," *Journal of Network and Computer Applications*, vol. 34, no. 5, pp. 1648–1659, 2011.

[24] C.-S. Feng, J. Yang, Z.-G. Qin, D. Yuan, and H.-R. Cheng, "Modeling and analysis of passive worm propagation in the P2P file-sharing network," *Simulation Modelling Practice and Theory*, vol. 51, pp. 87–99, 2015.

[25] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The Eigentrust algorithm for reputation management in P2P networks," in *Proceedings of the Twelfth International Conference on World Wide Web*, p. 640, Budapest, Hungary, May 2003.

[26] L. Xiong and L. Liu, "PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 843–857, 2004.

[27] R. Zhou and K. Hwang, "PowerTrust: a robust and scalable reputation system for trusted peer-to-peer computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 4, pp. 460–473, 2007.

[28] I. Stirling, W. Calvert, and C. Spencer, "Evidence of stereotyped underwater vocalizations of male Atlantic walruses (*Odobenus rosmarus rosmarus*)," *Canadian Journal of Zoology*, vol. 65, no. 9, pp. 2311–2321, 1987.

[29] L. Cai and R. Rojas-Cessa, "Mitigation of malware proliferation in P2P networks using double-layer dynamic trust (DDT) management scheme," in *Proceedings of the IEEE Sarnoff Symposium*, pp. 1–5, Princeton, NJ, USA, April 2009.

[30] D. Minoli and B. Occhiogrosso, "Blockchain mechanisms for IoT security," *Internet of Things*, vol. 1–2, pp. 1–13, 2018.

[31] M. A. Khan and K. Salah, "IoT security: review, blockchain solutions, and open challenges," *Future Generation Computer Systems*, vol. 82, pp. 395–411, 2018.
[32] Y. Qian, Y. Jiang, J. Chen et al., "Towards decentralized IoT security enhancement: a blockchain approach," *Computers & Electrical Engineering*, vol. 72, pp. 266–273, 2018.
[33] I. Makhdoom, M. Abolhasan, H. Abbas, and W. Ni, "Blockchain's adoption in IoT: the challenges, and a way forward," *Journal of Network and Computer Applications*, vol. 125, pp. 251–279, 2019.
[34] M. Ammar, G. Russello, and B. Crispo, "Internet of things: a survey on the security of IoT frameworks," *Journal of Information Security and Applications*, vol. 38, pp. 8–27, 2018.
[35] A. K. Das, S. Zeadally, and D. He, "Taxonomy and analysis of security protocols for Internet of things," *Future Generation Computer Systems*, vol. 89, pp. 110–125, 2018.
[36] A. D. Wood and J. A. Stankovic, "Denial of service in sensor networks," *Computer*, vol. 35, no. 10, pp. 54–62, 2002.
[37] J. P. Wack, K. Cutler, and J. Pole, *Guidelines on Firewalls and Firewall Policy*, Booz Allen Hamilton Inc, McLean, VA, USA, 2002.
[38] K. A. Scarfone and P. Hoffman, *Guidelines on Firewalls and Firewall Policy*, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2009.
[39] R. P. Hinglaspure and B. R. Burghate, "Analysis of packet filtering technology in computer network security," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 4, pp. 1302–1927, 2014.
[40] O. Kirch and T. Dawson, *Linux Network Administrator's Guide*, O'Reilly Media, Newton, MA, USA, 2nd edition, 2000.
[41] J. Vos and W. Konijnenberg, "IPFWADM: Linux firewall facilities for kernel-level packet filtering," in *Proceedings of the NLUUG Spring Conference*, Amsterdam, Netherlands, May 1996.
[42] R. Russell, Linux IPCHAINS-HOWTO, 2000.
[43] J. Stanger and P. T. Lane, "Chapter 9—implementing a firewall with ipchains and iptables," in *Hack Proofing Linux*, pp. 445–506, Syngress, Burlington, MA, USA, 2001.
[44] P. Russell, Netfilter: Firewalling, NAT, and Packet Managing for Linux, 2000.
[45] B. Sharma and K. Bajaj, "Packet filtering using IP tables in Linux," *International Journal of Computer Science Issues*, vol. 8, pp. 320–325, 2011.
[46] A. Alemayhu, "Moving from iptables to nftables," 2018, https://nftables.org.
[47] J. Corbet, "BPF comes to firewalls," 2018, https://lwn.net/.
[48] K. Ingham and S. Forrest, "A history and survey of network firewalls," Technical Report 2002-37, University of New Mexico, Albuquerque, NM, USA, 2002.
[49] R. Antonello, S. Fernandes, C. Kamienski et al., "Deep packet inspection tools and techniques in commodity platforms: challenges and trends," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1863–1878, 2012.
[50] T. Bujlow, V. Carela-Español, and P. Barlet-Ros, "Independent comparison of popular DPI tools for traffic classification," *Computer Networks*, vol. 76, pp. 75–89, 2015.
[51] Netresec A. B., CapLoader, 2018.
[52] Ntop Team, Say Hello to nDPI 2.0, 2017.
[53] Netresec, "Publicly available PCAP files," 2018, http://www.netresec.com.
[54] B. Cruz, D. Ruano-Ordás, and J. R. Mendez, FilterTLK, 2019.
[55] Oracle, *Trail: Creating a GUI with JFC/Swing*, Oracle, Redwood City, CA, USA, 2017.
[56] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo, The Programming Language Lua, Lua.Org, Brazil, ISBN: 8590379868, 4th edition, 2016.
[57] H. Kaplan, Lua Dissectors, 2015.
[58] W. Shotts, *LinuxCommand: Dialog*, No Starch Press, San Francisco, CA, USA, 2000.
[59] The Tcpdump Group, Tcpdump and Libpcap, 2010.
[60] Netfilter Project, Ebtables, http://ebtables.netfilter.org.
[61] M. Parkour, Contagio—Malware Dump, 2014.
[62] B. Duncan, "A Source for Pcap files and Malware Samples," Malware-Traffic-Analysis. 2018.
[63] B. Dolan-Gavitt, GTISK PANDA Malrec—PCAP Files from Malware Samples Run in PANDA, 2018.
[64] S. Renfo and B. Guyton, MergeCap: merges two or more capture files into one 2018, https://www.wireshark.org/docs/man-pages/mergecap.html.
[65] A. Fanjul, LG SuperSign RCE, 2018.
[66] Common Vulnerabilities and Exposures, CVE-2018-17173, 2018, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17173.
[67] Common Vulnerabilities and Exposures, CVE-2018-15887, 2018, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15887.
[68] Raspberry P. I., Raspberry Pi 2 Model B, 2018.
[69] Apache Software Foundation, *Apache HTTP Server Benchmarking Tool: Apache HTTP Server Version 2.4*, Apache Software Foundation, Forest Hill, MD, USA, 2018.
[70] O. Tange, GNU Parallel 2018, 2018.
[71] L. Zhao, A. Shimae, and H. Nagamochi, "Linear-tree rule structure for firewall optimization," in *Proceedings of the Sixth {IASTED} International Conference on Communications, Internet, and Information Technology*, pp. 67–72, Banff, Canada, July 2007.
[72] L. Defert, Iptables-optimize, 2014.
[73] VizSec Group, "VizSec ciber security datasets," in *Proceedings of the IEEE Symposium on Visualization for Cyber Security*, Berlin, Germany, June 2019, https://vizsec.org/.