Linux Network Scripting with Lua

Lourival Vieira Neto, Victor Nogueira, Ana Lúcia de Moura and Roberto Ierusalimschy
Ring-0 Networks and Departamento de Informática, PUC-Rio
Rio de Janeiro, Brazil
lourival.neto@ring-0.io, victor.nogueira@ring-0.io, analuciadm@gmail.com, roberto@inf.puc-rio.br

Abstract

Scriptable operating system is a design based on the idea that operating systems should allow users to write scripts to tailor the system to their needs. In this work we discuss how Lunatik, our kernel-scripting framework for Linux, addresses the challenges of scripting the kernel in terms of correctness, isolation, and liveness. We then discuss NFLua, which allows extending Netfilter with kernel-level Lua scripting, and introduce XPDLua, which extends XDP. Due to its integration to XDP, XDPLua allows Lua scripts to filter traffic before it reaches the network stack. In this case, Lua works as a scripting language to eBPF, allowing eBPF programs to call Lua scripts and Lua scripts to call eBPF helpers. To evaluate our scripting framework, we developed two packet filtering applications with XDPLua, and compared the execution of one of this filters implemented with eBPF, NFLua and XDPLua.

Keywords

Lua, Kernel Scripting, Packet Filtering

1 Introduction

Lunatik is an extension framework for Linux that combines the approaches of *Extensible Operating Systems* [17] and *scripting languages* [28]. This combined approach, named *Kernel Scripting* [33], advocates that common OS kernels can be dynamically extended by using a high-level scripting language. Lunatik [35] is a programming and execution environment that offers two main functionalities for extending Linux kernel subsystems: it provides support to kernel developers for making their subsystems scriptable and allows users to dynamically load and run Lua scripts in the kernel.

Lua [15, 18] is an extensible extension programming language that has been specifically designed as an *embedded* language, and thus can easily both extend and be extended by host programs [12, 13]. It is implemented as a regular C library with a well-defined API [14, 24], and has been widely used for scripting network monitoring and security tools, such as Wireshark [16], Nmap [4], Snort [5] and ModSecurity [30]. It has also been used in Snabb [31], a scripting framework for packet processing that uses the Data Plane Development Kit (DPDK). Since 2014, Lua is present in the NetBSD Operating System base as a kernel module [2], and has been used, among other applications, to script NetBSD's NPF packet filter [36, 38].

The Lunatik framework has already been used for scripting some Linux network subsystems such as Netfilter [3, 9], Sockets [32] and, more recently, eXpress Data Path (XDP) [27]. In order to make these subsystems scriptable, we needed to develop *language bindings*, that is, a glue code that binds the host program —the kernel network subsystems, in this case— to the Lua interpreter. These bindings allow the kernel subsystems to call Lua functions like, for instance, a user-defined Lua filtering function that processes a packet and returns a verdict. They also provide features to Lua scripts such as abstractions that represent packets and access to network internal mechanisms.

In this paper, we discuss two special bindings for Linux network subsystems: NFLua, for Netfilter [25], and XDPLua, for XDP [11]. Although both bindings are targeted to allow users to develop custom packet filtering techniques, XDPLua, an evolution of NFLua, allows Lua scripts to filter traffic before it reaches the network stack and provides a significant better performance.

The rest of this paper is organized as follows. Section 2 discusses the challenges of scripting the kernel and how to address them using Lua's support for sandboxing scripts in an isolated, restricted and safe execution environment. Sections 3 and 4 discuss respectively NFLua, the Netfilter binding for Lunatik, and XDPLua, the binding for XDP. Section 5 presents some network scripting applications developed with XDPLua, and compare the execution of access control filters developed with XDPLua, NFLua and eBPF. Finally, Section 6 presents our final remarks.

2 Sandboxing Kernel Scripts

Like conventional kernel extensions, kernel scripts should not introduce malfunctioning to the system; therefore, they are subject to the same issues concerning these extensions, such as correctness, liveness and isolation. More specifically, kernel scripts should not crash or corrupt the system, run indefinitely or block an execution flow, or corrupt resources owned by other users.

However, differently from conventional extensions, kernel scripts are not supposed to address such issues by themselves, due to the higher-level nature of scripting. Instead, their execution environment is responsible for guaranteeing that they do not compromise the system. With the help of programming support provided by the Lua language itself, our ker-

nel scripting framework, Lunatik, implements a safe executing environment by running kernel scripts in a sandbox, described in this section.

Lua scripts do not allocate or have direct access to memory. They can only handle memory in the form of Lua data types, such as strings, tables, and userdata. Because Lua has dynamic memory management, all data objects are allocated internally by the Lua VM, and are subject to a garbage collector. This frees the script developer from handling memory management issues, typically a source of correctness problems.

Lua scripts are also restricted to the APIs exposed to a given Lua execution state (an isolated execution environment that corresponds to an instance of the Lua interpreter). Lua states are created empty: only language operators are initially available. To allow a script to access additional facilities — including those provided by Lua standard libraries— we must explicitly load them into the state inside which the script is running. Lua standard libraries offer basic facilities such as string and table manipulation, mathematical operations and coroutine primitives. For most scenarios, we load them just after the state creation. Both libraries and independent registered functions can be selected per execution state, according to the extension purpose.

Besides Lua standard libraries, we also need to load specific libraries to extend Lua with new features, suited for the purpose of the scripting extensions. We call these libraries *extension bindings*, because they extend Lua with new functionalities. We also need to create modules or change the kernel to make it possible to call Lua scripts. We call these modules (or patches) *embedding bindings*, because they allow Lua to be embedded in the kernel. As an example, to script Netfilter and XDP we have two embedding bindings: NFLua and XDPLua, respectively. The former was implemented as a loadable kernel module and the latter as a kernel patch, due to the lack of support of extensions in XDP.

Because Lua does not permit scripts to directly access memory, we have also created an extension binding, called Luadata [34], that allows Lua scripts to access memory external to its execution state. The API provided by Luadata allows embedding bindings, such as NFLua and XDPLua, to expose memory (e.g., socket buffers) to be handled safely by Lua scripts. In this case, both the extension binding (Luadata) and the embedding bindings (NFLua and XDPLua) are responsible for ensuring correctness. Luadata is responsible for checking boundaries on every access to the exposed objects. The embedding bindings specify the boundaries to be checked, and are responsible for attaching and detaching the exposed data objects to and from the Lua state, because this memory is managed by the kernel, and not by Lua.

Although Lua kernel scripts do not have direct access to memory, they could potentially corrupt the system by allocating too many resources. To address this issue, Lua allows the customization of the memory allocator of an execution state. NFLua, for instance, uses this facility to cap the amount of memory used by each Lua state to prevent scripts from allocating memory indefinitely.

Because Lua scripts run in a single-threaded execution environment, Lua does not provide primitives for synchronization, such as mutexes. Instead, Lua provides coroutines [23], which support collaborative multitasking. Lua scripts thus cannot explicitly lock the kernel flow that executes them. However, they could potentially corrupt the system by running for too long. To address this issue, Lua can interrupt a script after running a specific amount of instructions. XDPLua uses this facility to cap the amount of instructions executed inside each Lua state, preventing scripts from running indefinitely.

To allow multitasking, Lunatik permits the creation of multiple Lua execution states inside the kernel. These multiple states are completely isolated, not sharing memory. However, it is possible to extend Lua with communication mechanisms for sharing data between execution states. LuaRCU [20] is an extension binding that allows Lua execution states to safely share memory using the RCU (read-copy-update) support provided in the Linux kernel [19].

The sandboxed environment is also responsible for guaranteeing that only privileged users can extend the kernel using scripts. Both NFLua and XDPLua allow access only for users with network-administration capabilities (`CAP_NET_ADMIN`). They use Netlink sockets [1] to communicate with user space and validate user capabilities on every access.

3 NFLua

Netfilter [25] is a packet filtering framework provided by the Linux kernel. It implements a set of hooks at well-defined points in the packet traversal of the network stack, allowing registered callback functions to receive and act upon every packet that reaches their associated hooks. The callback functions are defined by loadable kernel modules that dynamically extend Netfilter in order to set up custom filtering strategies. Implemented as an extension module to Netfilter, NFLua [3] provides a kernel-scripting framework for packet filtering using Lua [1].

```
function checkuseragent(pkt)
  -- extracts User-Agent HTTP header
  local pattern = "User%-Agent:%s(.-)\r\n"
  local useragent = string.match(pkt, pattern)

  return blocklist[useragent]
end
```

Figure 1: Inspecting the *User-Agent* HTTP header in Lua

An example of a Layer 7 filtering strategy that can be easily implemented with NFLua is blocking connections based on the HTTP *User-Agent* header. In this example, the Lua function `checkuseragent`, shown in Figure 1, is responsible for extracting the header from the HTTP request and verifying its presence in a blocklist. Pattern matching facilities provided by Lua greatly simplifies this task.

[1] NFLua was inspired by NPFLua, which was developed for NetBSD's NPF.

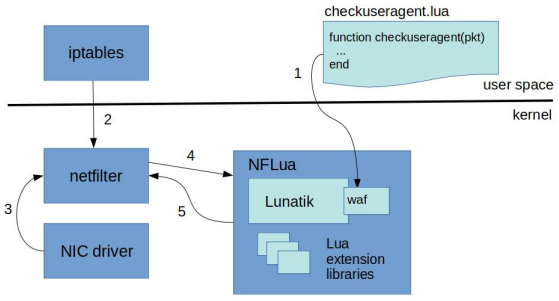

Figure 2: Setting up a filtering strategy with NFLua

Figure 2 illustrates how this filtering strategy can be set up. We first create a Lua execution state in Lunatik and load the function `checkuseragent` into it (1). For creating Lua states and loading scripts into the kernel, we use the `nfluactl` tool, which communicates with NFLua through a Netlink socket. The following commands achieve this task:

```
nfluactl create waf
nfluactl execute checkuseragent.lua
```

Next (2), we set an Iptables rule to make Netfilter send every packet destined to TCP port 80 to NFLua:

```
iptables -A INPUT -m TCP --dport 80 -m lua
  --state waf --function checkuseragent -j REJECT
```

After the filtering strategy has been set up, whenever an HTTP packet is received (3), Netfilter will send it to a callback function defined by NFLua (4). This callback function will then call the function `checkuseragent`, running in state `waf`, passing the captured packet to it. If a match is found for the packet, Netfilter is instructed to terminate the connection (5).

Besides applying Lua-defined matching rules to decide whether to accept or discard a packet, NFLua can also use Lua scripts to define the action to be performed when a match is found (the Iptables *target*). This facility makes it possible to implement more advanced logic, such as modifying the packet content or stealing the packet for later use.

Finally, NFLua also allows scripts to transfer data to user space, by means of Netlink sockets. This facility can be useful for tools and applications that need to capture data from network traffic.

4 XDPLua

XDP is a mechanism that provides a safe execution environment for custom packet processing applications [11]. To allow the execution of custom applications inside the data path, XDP lets users dynamically load eBPF programs into the kernel. However, because kernel space is a sensitive context, it is essential to guarantee that these programs will not compromise the system. To achieve this guarantee, an in-kernel verifier analyses the eBPF programs at load time, and decides whether they should be allowed to execute inside XDP. Although the restrictions imposed by this verifier are important to secure system stability, they also limit the expressiveness of eBPF programs, making it hard for programmers to create more complex applications [21].

In order to provide developers with a more expressive environment, we developed XDPLua, an extension to XDP that allows Lua scripts to be loaded and executed inside the data path. Parsing Layer 7, discussed in the previous section, is an example of a task that can easily be implemented with Lua, but quite difficult to implement with eBPF.

Although Lua's expressiveness can be a valuable addition to XDP, its flexibility comes with a price in performance, specially when compared to compiled languages. However, in environments where performance matters, Lua is typically used for scripting a high-performance *system* language, allowing developers to implement the heavy work in this language, and use Lua for tasks where expressiveness and flexibility are advantageous.

In XDPLua, what we essentially do is to script eBPF —the system language— with Lua. By means of eBPF helpers —essentially C functions—, eBPF programs can call Lua scripts and Lua scripts can have access to kernel functionalities. Scripting eBPF with Lua makes it possible to take advantage of Lua's flexibility and expressiveness while still being able to make good use of eBPF's high performance, combining the best of both languages.

The helpers we have created so far are mostly wrappers to the Lua C API, and are responsible for assuring that the eBPF program cannot compromise the system when invoking a Lua script.

To better illustrate how we can use XDPLua, we will follow the same example implemented with NFLua in the previous section: blocking connections based on the HTTP *User-Agent* header. First, we need to create an eBPF program that calls the Lua function `checkuseragent` —presented in the previous section— and drops the packet depending on the function's returned verdict. In order to call the Lua function and retrieve its result, it uses some helper functions, prefixed with "`bpf_lua_`".

As with NFLua, we need to have an execution state where we can load the Lua function. However, in XDPLua the user does not create Lua states because they are created by XDPLua itself, one per available CPU. By doing so, we can take advantage of the parallelism of multi-core architectures, which helps with performance. If it is necessary to persist data among different executions of a Lua function, the extension binding LuaRCU [19], previously mentioned, can provide safe support for sharing data among the Lua states.

Figure 3 illustrates how the filtering strategy can be set up. First we load the eBPF program, associating it to a network interface (1). Next, we load the desired Lua function (`checkuseragent`) into all Lua states created by XDPLua (2). When an incoming packet arrives at the network interface, it is passed to XDP, which executes the eBPF program associated to that interface (3). The eBPF program then calls the Lua function `checkuseragent` (4). The Lua state used to execute this function is the one associated to the CPU which executes the eBPF program. Based on the value returned by the Lua function, the eBPF program decides what to do with the packet (5): accept it or drop it.

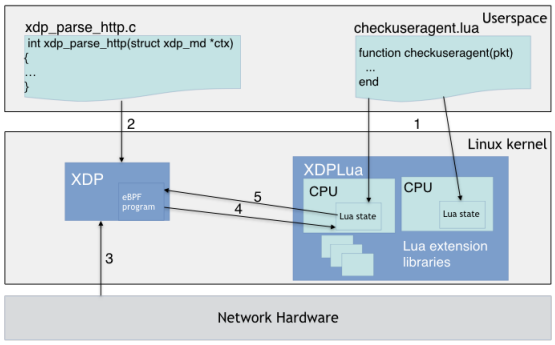

Figure 3: Setting up a filtering strategy with XDPLua

5 Network scripting with XDPLua

In order to evaluate our framework, we developed two network scripting applications with XDPLua. Both applications require computations that are quite difficult to implement with eBPF, but easily developed with Lua. By combining the two languages, we were able to have expressiveness, where adequate, and performance, where indispensable.

JavaScript Challenge

JavaScript Challenge, or JavaScript Authentication, is a DDoS (Distributed Denial-of-Service) mitigation technique that identifies and blocks malicious bots by embedding JavaScript code in the response sent from an HTTP server to its clients [22]. The basic idea is that typical attack tools do not incorporate a JavaScript engine, and thus cannot perform the requested computation.

We implemented this technique by integrating an Nginx server [26] with XDPLua. The JavaScript code that we send to the server's clients tells them to set a specific cookie (`__xdp`) with a random value. The cookie value associated to each client is sent to XDPLua, allowing Lua code to authenticate further HTTP requests sent by those clients.

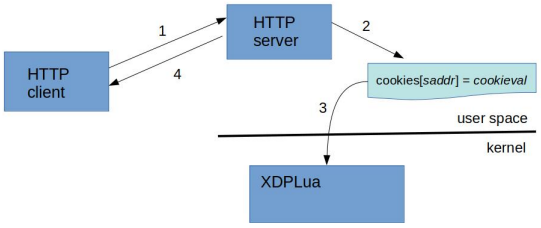

Figure 4: Setting up the JavaScript challenge

Figure 4 illustrates what happens when the HTTP server receives the first HTTP request from a given client. When the server receives the request (1), it generates the random value for the client's cookie, creates a Lua script (2) and injects it into XDPLua (3). When this script is executed inside XDPLua, it creates an entry in the Lua table `cookies`, associating the client's IP address (`saddr`) to its cookie value (`cookieval`). Finally, the HTTP server embeds the JavaScript code containing the generated cookie in the response, and sends it to the client (4).

```
function checkcookie(pkt, saddr)
  -- checks if challenge is not set yet (first request)
  if not cookies[saddr] then
    return true
  end

  -- extracts __xdp cookie
  local pattern = "Cookie:%s*=__xdp=(%d+)%s*"
  local cookieval = string.match(tostring(pkt), pattern)

  -- checks cookie's value
  return cookies[saddr] == cookieval
end
```

Figure 5: Validating the JavaScript Challenge

A combination of eBPF and Lua code is responsible for detecting and blocking malicious clients. An eBPF program loaded into XDP analyses the lower layers of each received packet. If the transport layer is TCP, and the destination port is 80 —i.e., it is an HTTP request—, the eBPF program invokes Lua function `checkcookie`, presented in Figure 5, which extracts the cookie from the request and checks if it is correct. If this function fails to authenticate the request, the eBPF program inserts its IP address in a blocklist, and will drop any further packets sent by this client.

Implementing the JavaScript Challenge with a combination of eBPF and Lua code is a good example of how to benefit from the strengths of both languages. Authenticating HTTP requests based on a cookie value is a trivial task to implement in Lua. On the other hand, it would be harder to implement with eBPF, due to the limitations imposed by the in-kernel code verifier and the lack of pattern-matching facilities. Once a malicious client is detected, eBPF code is responsible for blocking it, without further need to invoke Lua code.

Access Control

Controlling access to blocked hosts can be useful in several scenarios, such as limiting access time to a specific site, preventing access to inappropriate content, and protecting users from well-known malicious domains. Our next application implements this kind of control in a gateway that filters TLS (Transport Layer Security) outbound traffic, dropping messages that contain, in their SNI (Server Name Indication), a destination hostname that is registered in the blocklist.

Our filtering strategy, illustrated in Figure 6, was again implemented with a combination of eBPF and Lua code. To set up this strategy, we load a hostname blocklist —represented as a Lua table— and the Lua function `checksni` into all Lua states created by XDPLua (1 and 2). We also load the eBPF program into XDP (3). After that, all packets that arrive at the

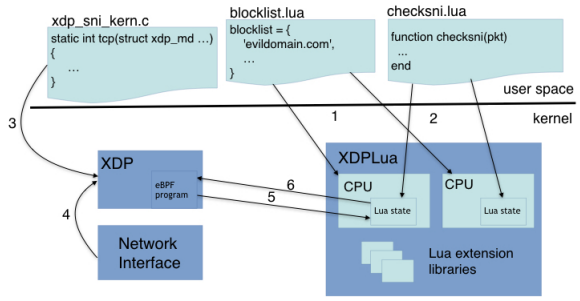

Figure 6: Controlling access to blocked hostnames

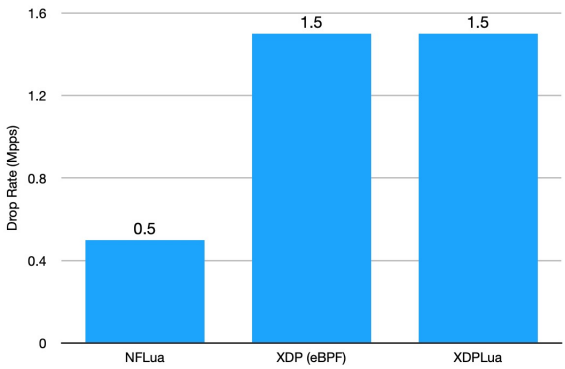

Figure 7: Access Control drop rate

network interface associated to the eBPF program are sent to this program (4). If the received packet is a TCP packet destined to port 443, the eBPF program sends it to Lua function `checksni` (5). If the packet contains a *client hello* message, the Lua function extracts its SNI and verifies whether it is registered in the blocklist. According to the verdict returned by this function, the eBPF program decides whether the packet should be dropped or not.

Because TLS messages have optional fields with variable sizes, extracting the SNI from a TLS message is a rather complex task even for Lua code, and specially difficult to implement using only eBPF. Although we were able to do it purely in eBPF, the result was 200 lines of a cumbersome C code, and much effort spent to circumvent the in-kernel verifier. In contrast, our Lua implementation has only 37 lines and a clearer code. Our pure eBPF implementation also has a major limitation because, due to the limitations imposed by the in-kernel code verifier, we were not able to make the eBPF program check the captured SNI against a blocklist. Thus, it can only block a single hostname.

Since we were able to implement our access control strategy with pure eBPF, and also with NFLua, we performed a preliminary benchmark to compare these implementations. To make it fair, all implementations blocked a specific hostname, including the one in XDPLua, and both XDPLua and NFLua implementations used the same Lua function to extract and check the SNI.

The goal of our benchmark was to compare packet drop rates and CPU consumption, and our test environment consisted of a client machine continuously sending TLS *client hello* messages to a server running each of our packet filtering implementations. All the messages sent were destined to the blocked hostname and had 583 bytes.

Both client and server were virtualized machines on AWS (Amazon Web Services) with an 8 core Intel Xeon Platinum 8124M CPU running at 3.00 GHz, 32 GB of RAM, one 10 Gbps Virtio network interface, and the Ubuntu Bionic Linux distribution. The kernel version was 5.3.0 in the client and 5.6.0 —modified to use XDPLua— in the server. For sending the TLS messages we used the traffic generator Trafgen [10]; for measuring CPU time we used Mpstat [7].

Figure 7 shows the observed drop rate in Mpps and Figure 8 shows the observed CPU usage. Both the XDPLua and the XDP (eBPF) implementations dropped packets at the same rate, 1.5 Mpps (7 Gbps), and consumed approximately 0.1% of CPU. The NFLua implementation dropped only one third of the rate achieved by XDPLua and XDP (eBPF), 0.5 Mpps (2.3 Gbps), and consumed approximately 500 times more CPU than these two implementations.

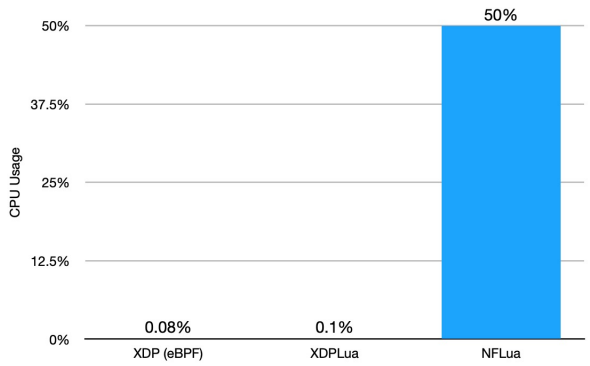

Figure 8: Access Control CPU usage

Because we were restricted to a virtualized network environment which we could not fully control, we could not generate traffic rates higher than 1.5 Mpps. Therefore, none of the tested implementations could be stressed to its limit. Despite this limitation, our benchmark supported our expectations that XDPLua performs considerably better than NFLua from both CPU and throughput perspectives. Although the observed results do not show a significant performance difference between the XDPLua and XDP (eBPF) implementations for *Access Control*, we expect the difference to be considerably more significant for higher network throughput or lower computational power. In this case, we can combine Lua and eBPF, instead of calling Lua to process every incoming

packet, as shown on our *JavaScript Challenge* implementation.

6 Final Remarks

This paper discussed how Lunatik, our framework for scripting Linux network subsystems, addresses issues concerning code injected into the kernel. It also presented NFLua and XDPLua, the bindings that permit extending Netfilter and XDP, and showed how these bindings allow users to implement advanced packet filtering techniques using the Lua programming language.

NFLua has been used by network operators for some years now, for implementing advanced cybersecurity and network monitoring features [39]. It is currently present in around 20 million home routers to protect and monitor over 500 million devices [8].

XDPLua was designed as the natural evolution of NFLua, using XDP instead of Netfilter, and applying lessons learned from NFLua's development, such as using separate execution states per CPU to avoid blocking the network processing flow. Unfortunately, differently from Netfilter, XDP does not support extensions as loadable kernel modules; the need to develop an out-of-tree binding made XDPLua considerably harder to implement.

XDPLua is currently used in the *Ring-0 Firewall* [29], also for implementing advanced cybersecurity and monitoring features for network operators. Instead of running inside home routers, it runs at the network edge. In February 2020, this firewall solution was deployed at the GoCache CDN [6], and since then it has been running in points of presence (PoP) with 10 Gbps bandwidth. In this scenario, we have observed that, during peaks of around 5.3 Gbps (3.7 Mpps), only up to 4% of CPU [2] was in use. As a comparison, NFLua has not been able to handle more than 1 Gbps traffic [37] without considerably penalizing the CPU, mostly because it runs in Netfilter hooks and does not support parallelism.

XDPLua was also designed to use Lua cooperatively with eBPF. Although not matching eBPF's performance, Lua adds the expressiveness and ease of use of a full-fledged and high-level programming language. XDPLua thus allows users to create sophisticated network extensions using a combination of Lua and eBPF, using eBPF for performance-critical operations, and Lua for more complex tasks.

Instead of limiting a language by removing Turing-completeness or imposing an in-kernel verifier, we apply regular sandboxing techniques using facilities provided by the language interpreter itself. This facilitates the overall user experience, avoiding complicated tooling without losing expressiveness in the extension language.

Acknowledgments

XDPLua was partially developed as a Google Summer of Code project, sponsored by LabLua. We thank Marcel Moura, Pedro Tammela and Matheus Izvekov for all their valuable contribution to the development of XDPLua and NFLua. We also thank LNCC (National Laboratory for Scientific Computation), CAPES (the Brazilian Agency for the Improvement of Higher Education) and GoCache for all given support.

[2] Intel Xeon Silver 4114, 8 cores, 16GB of RAM

References

[1] Ayuso, P. N.; Gasca, R. M.; and Lefevre, L. 2010. Communicating between the kernel and user-space in Linux using Netlink sockets. *Software: Practice and Experience* 40(9). https://perso.ens-lyon.fr/laurent.lefevre/pdf/JS2010_Neira_Gasca_Lefevre.pdf.

[2] Balmer, M. 2014. LUA(4) NetBSD Kernel Interfaces Manual. https://netbsd.gw.com/cgi-bin/man-cgi?lua+4+NetBSD-9.0.

[3] CUJO LLC. NFLua. https://github.com/cujoai/nflua.

[4] Donnelly, P. 2009. Nmap Script Engine: implementation and uses. Talk presented at Lua Workshop 2009, Rio de Janeiro, Brazil. https://www.lua.org/wshop09/NSE-Lua-Workshop.odp.

[5] Etienne, L. 2009. Malicious traffic detection in local networks with snort. Technical report, EPFL.

[6] GoCache. GoCache next-gen CDN. https://www.gocache.com.br/.

[7] Godard, S. mpstat - Report processors related statistics. http://sebastien.godard.pagesperso-orange.fr/man_mpstat.html Accessed in: June 17th 2020.

[8] Goemaere, P. 2019. The Evolution of Network Virtualization In The Home. *The NCTA Technical Papers*.

[9] Graf, A. 2010. PacketScript - a Lua Scripting Engine for in-Kernel Packet Processing. Master's thesis, Computer Science Department, University of Basel. https://cn.dmi.unibas.ch/pub/doc/2010-msthGraf.pdf.

[10] Helvey, E. L. 1998. *Trafgen: An Efficient Approach to Statistically Accurate Artificial Network Traffic Generation*. Ph.D. Dissertation, Ohio University.

[11] Hoiland-Jorgensen, T.; Brouer, J. D.; Borkmann, D.; Fastabend, J.; Herbert, T.; Ahern, D.; and Miller, D. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'18)*, 54–66. Association for Computing Machinery (ACM), New York, USA. https://dl.acm.org/doi/abs/10.1145/3281411.3281443.

[12] Ierusalimschy, R.; de Figueiredo, L.; and Celes Filho, W. 1996. Lua—an extensible extension language. *Software: Practice and Experience* 26(6):635–652.

[13] Ierusalimschy, R.; de Figueiredo, L.; and Celes, W. 2007. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, 2–26. ACM.

[14] Ierusalimschy, R.; de Figueiredo, L. H.; and Celes, W. 2011. Passing a language through the eye of a needle. *Communications of the ACM* 54(7):38–43.

[15] Ierusalimschy, R. 2016. *Programming in Lua*. Lua.org, Fourth edition.

[16] Lamping, U., and Warnicke, E. 2004. Wireshark user's guide. *Interface* 4(6).

[17] Lampson, B. 1969. On reliable and extendable operating systems. In *Proceedings of the Second NATO Conference on Techiniques in Software Engineering*.

[18] Lua.org. The Programming Language Lua. http://www.lua.org.

[19] McKenney, P. 2007. What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[20] Messias, C. L. RCU Binding for Lunatik.

[21] Miano, S.; Bertrone, M.; Risso, F.; Tumolo, M.; and Bernal, M. V. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *Proceedings of the 19th International Conference on High Performance Switching and Routing (HPSR)*, 1–8. IEEE. https://ieeexplore.ieee.org/document/8850758.

[22] Miu, T.; Hui, A.; Lee, W.; Luo, D.; Chung, A.; and Wong, J. 2013. Universal DDoS mitigation bypass. *Black Hat USA*. https://media.blackhat.com/us-13/US-13-Lee-Universal-DDoS-Mitigation-Bypass-WP.pdf.

[23] Moura, A. L. d., and Ierusalimschy, R. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31(2). https://doi.org/10.1145/1462166.1462167.

[24] Muhammad, H., and Ierusalimschy, R. 2007. C APIs in Extension and Extensible Languages. *Journal of Universal Computer Science* 13(6):839–853.

[25] netfilter.org. Netfilter: firewalling, NAT, and packet mangling for Linux. http://www.netfilter.org.

[26] nginx.org. NGINX: High Performance Load Balancing, Web Server and Reverse Proxy. https://nginx.org/en/.

[27] Nogueira, V. B. XDPLua. https://victornogueirario.github.io/xdplua.

[28] Ousterhout, J. 1998. Scripting: Higher-level programming for the 21st century. *IEEE Computer* 31(3):23–30.

[29] Ring-0. Ring-0 Firewall. https://ring-0.io.

[30] Ristic, I. 2010. *ModSecurity Handbook*. Feisty Duck.

[31] Scholz, D. 2016. Diving into snabb. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 31.

[32] Tammela, P. ULP Lua. https://github.com/luainkernel/ulp-lua.

[33] Vieira Neto, L.; Ierusalimschy, R.; de Moura, A. L.; and Balmer, M. 2014. Scriptable Operating Systems with Lua. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS'14)*, 2–10. Association for Computing Machinery (ACM), New York, USA. https://dl.acm.org/doi/proceedings/10.1145/2661088.

[34] Vieira Neto, L. Lua data library. https://github.com/lneto/luadata Accesed in: March 5th 2020.

[35] Vieira Neto, L. Lunatik: Lua in Kernel. https://github.com/luainkernel/lunatik.

[36] Vieira Neto, L. 2014. NPF Scripting with Lua. Talk presented at EuroBSDCon 2014, Sofia, Bulgaria. http://netbsd.org/ lneto/eurobsdcon14.pdf.

[37] Vieira Neto, L. 2017. Safe Browsing using Lua. Talk presented at Lua Workshop 2017, San Francisco, USA. https://www.lua.org/wshop17/Lourival.pdf.

[38] von Dollen, A. 2018. Fast, Flexible Packet Filtering in NetBSD using Lua Kernel Scripts. Talk presented at EuroBSDCon 2018, Bucharest, Romania. https://2018.eurobsdcon.org/static/slides/Fast,

[39] Wheelock, I., and Cheevers, C. 2019. 2019 Virtualized CPE Services Have Finally Arrived Via Service Delivery Platforms. *The NCTA Technical Papers*.