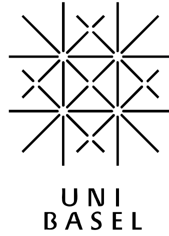


University of Basel

Computer Science Department

Master Thesis

PacketScript – A Lua scripting engine for in-kernel packet processing

Author:
André Graf

Thesis Advisors:
Prof. Dr. Christian Tschudin
Dr. Christophe Jelger

Monday 26$^{\text{th}}$ July, 2010

Abstract

Computer network researchers, telecommunication engineers, system administrators, and kernel hackers are extending Linux Netfilter to develop new networking protocols, fine-tune the networking setup, or for debugging purposes. Extending Linux Netfilter requires solid C programming skills and a good understanding of the Linux kernel and its network stack. This know-how is critical, since a small programming mistake can already tear-down the whole operating system. For this reason, we think that Linux Netfilter is not well suited for rapid prototyping. To overcome this problem we developed PacketScript; PacketScript embeds the Lua virtual machine in Linux Netfilter and provides a fully scriptable interface for analyzing and modifying network packets in an object oriented way using Lua. Moreover, PacketScript is not constrained to a specific networking protocol or network layer. Experiments showed that the additional overhead generated by PacketScript is small enough to compete with common Linux tools. With PacketScript, a rapid development process on an integral part of the Linux kernel becomes possible. Furthermore, the development of new network functionality could be shifted much earlier from a simulation environment to a productive environment, hence being able to have a working prototype available at an earlier stage in the development process.

Acknowledgments

It is a pleasure to thank those who made this thesis possible. At first, I would like to thank Prof. Dr. Christian Tschudin for his encouragement and making this thesis possible. I am also grateful to all members of the Computer Networks Research Group of the University of Basel that supported me during this work. Especially I am grateful to my supervisor Dr. Christophe Jelger for many insightful conversations during the development of the ideas in this thesis, and for helpful comments on the documentation. Furthermore, I would like to thank my friends Christoph Jud, Florian Zeller and Filip Brinkmann for all the discussions and ideas. Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Contents

Abstract i

Acknowledgments ii

1 Introduction 1

1.1 Problem Description . . . 2

1.2 Related Work . . . 2

1.3 Thesis Outline . . . 3

2 Background 4

2.1 Netfilter . . . 4

2.1.1 Userspace Plugin . . . 6

2.1.2 Linux Kernel Module . . . 6

2.2 Lua . . . 7

2.2.1 Metatables . . . 8

2.2.2 C API . . . 8

3 PacketScript 10

3.1 Specification . . . 10

3.1.1 Functionality . . . 11

3.1.2 External Interfaces . . . 11

3.1.3 Performance . . . 11

3.1.4 Nonfunctional Requirements . . . 12

3.1.5 Design Constraints . . . 12

3.2 Architecture . . . 12

3.2.1 Iptables Userspace Plugin . . . 14

3.2.2 Netfilter Extension . . . 14
3.2.3 Software Interrupt Context vs. Process Context . . . 15
3.3 Object Oriented Packet Scripting . . . 20
3.3.1 Generation of Protocol Classes . . . 21
3.3.2 Field Modifiers . . . 23
3.4 Protocol Buffers . . . 24
3.4.1 The Raw Approach . . . 24
3.4.2 Structured Packet Access . . . 26
3.5 Dynamic Protocol Buffers . . . 28
3.6 Byte Arrays in Lua . . . 30
3.6.1 The *byte_array* object . . . 30
3.6.2 The Bytes Library . . . 31
3.7 The Netfilter Library . . . 31
3.7.1 Deferring Work . . . 32
3.7.2 Sending Packets . . . 32

4 Experiments **34**
4.1 Network Address Translation . . . 34
4.2 Application Level Packet Cache . . . 38
4.2.1 TFTP Cache . . . 38
4.2.2 HTTP Cache . . . 39
4.3 Discussion / Conclusions . . . 39
4.3.1 NAT Experiment . . . 39
4.3.2 Cache Experiment . . . 42

5 Conclusion **45**
5.1 Future Work . . . 45
5.2 Implications of Research . . . 47

Bibliography **49**

Chapter 1

Introduction

This thesis describes the development of PacketScript, a framework that enables to filter and to manipulate network packets using the Lua scripting environment inside the Linux kernel. Filtering and manipulating network packets are important building blocks of today's computer networks. These are used to set up firewalls and to connect different computer networks using NAT (Network Address Translation). For Linux based computers, Netfilter [1] provides a modular extensible system inside the Linux kernel for intercepting and manipulating network packets. Various disciplines in Linux network programming such as the development of new network protocols extend Netfilter for debugging and testing. However, Linux kernel development is a difficult process, especially when it comes to testing and debugging of the new kernel's functionality.

In recent years, there has been a paradigm shift towards higher level programming languages, whereas scripting languages gained significance. Typical characteristics of scripting languages are dynamic typing, automatic type conversions, and late binding as they are interpreted or just-in-time compiled. These features provide a perfect environment for a rapid development process. Another very interesting feature of scripting languages is that some are designed to be embedded in other applications, thus turning an otherwise static compiled application into a more dynamic piece of software. The Lua scripting environment [2] provides such an embeddable language, which is well known for its simplicity and the small memory footprint. Furthermore, it provides a friendly syntax, which is easy to learn. Compared to the ideas of scripting languages, Netfilter is a rather static environment. Changes in the source code require a recompilation of the Netfilter module, or even worse the recompilation of the whole Linux kernel. Having a scripting facility placed in Netfilter would

be a valuable feature for computer scientists, system administrators and kernel developers themselves, who may use it for debugging purposes and rapid prototyping.

1.1 Problem Description

In this thesis we will answer the question of how Lua can be embedded in Netfilter and show how Lua can be used inside Netfilter. More specifically we show how to load and unload Lua scripts, as well as how a Lua script accesses and modifies the different protocol headers and their fields. In order to demonstrate our design, we have successfully implemented a prototype of PacketScript.

1.2 Related Work

Kernelspace Lua. There are currently two projects focussing on porting Lua to the Linux kernel. Luak [3] aims at extending the kernel by loading a Linux kernel module. The project mainly contains one diff-file, which is used to patch the Lua sources. The patch is quite simple, thus basically patching function calls not being available within the Linux kernel. A more elaborate project is Lunatik [4], which extends the kernel by patching parts of the kernel, for example adding a new system call. It is striking that the project itself is a master thesis at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), the birthplace of Lua, where the leading architect of Lua, Roberto Ierusalimschy, is an associate professor of informatics. Even though both projects share the same objectives, we decided to use parts of Lunatik for PacketScript. There are several reasons for this decision, such as for example the support for failure handling. Furthermore, the project is actively used and there is a published paper for Lunatik [5]. Additionally, the Lunatik project will be ported to the NetBSD kernel as a *Google Summer of Code 2010* project, sponsored by the NetBSD Foundation [6].

Network Prototyping. There has been a number of studies on network prototyping. However, most solutions use a simulation-based approach to prototype new network functionality. The problem of moving a simulated scenario to a real scenario is addressed by a few network simulators, which try to reuse the network code employed for simulation also for the "real" system being developed. This is normally done by virtualizing parts of a system kernel and then simulating the underlying network functionality. Such an approach is used

by Entrapid [7], the Harvard TCP/IP network simulator [8], and Dummynet [9]. Another approach is taken by Nsclick [10], embedding the Click Modular Router [11] inside the ns-2 network simulator [12]. Another approach is considered by Alpine [13], providing a network stack in user space that can be used for easier network protocol development. A related motivation was the origin of JChannels [14] enabling the rapid prototyping of network protocols for the Java Virtual Machine. All these projects share the same idea of virtualizing parts of the system kernel in order to enable a rapid prototyping process. Because our approach works with the Lua scripting language, the scripts also run in a virtualized runtime environment that is, unlike for the other approaches, located inside the kernel space. A similar approach to ours is taken by GateScript [15] introducing a scripting language for expressing packet processing logic. Even tough the concepts are very similar, there are significant differences regarding the realization, for instance the usage of user space processes. There is also other work done, focusing on either one particular system platform, a special network protocol, a selected network layer, or a combination of all of them.

1.3 Thesis Outline

Chapter 1 sets the scope of the master thesis as well as points the reader to related work. In Chapter 2 we provide an overview of the Lua scripting environment and the development of Netfilter modules. Our main contribution the design and implementation of the PacketScript prototype is described in Chapter 3. This prototype is then used for performance evaluations, which are described and discussed in Chapter 4. Finally, in Chapter 5 we conclude this report.

Chapter 2

Background

In this chapter we provide background information about the technologies used in this project. While there are several books, papers, and presentations explaining these technologies in-depth, we provide some general information and point the reader to the different references for further reading. Section 2.1 shortly describes the Netfilter framework as well as how it can be extended. Section 2.2 introduces the Lua scripting environment.

2.1 Netfilter

Netfilter [1] is the component of the Linux kernel that is used when the network traffic needs to be inspected and/or manipulated. More specifically, Netfilter inserts five hooks into the networking stack (see Figure 2.1):

- **PREROUTING:** All packets traverse this hook. It is called before any routing decision is made, but after all IP header sanity checks have succeeded. Typically, Port Address Translation (PAT), the redirection of packets, as well as Destination Network Address Translation (DNAT) are implemented in the PREROUTING hook.

- **INPUT:** All incoming packets that are destined to the local machine pass this hook. This is the last hook traversed by incoming packets.

- **FORWARD:** All packets that are not destined to the local machine traverse this hook. This hook is typically used for implementing firewalls.

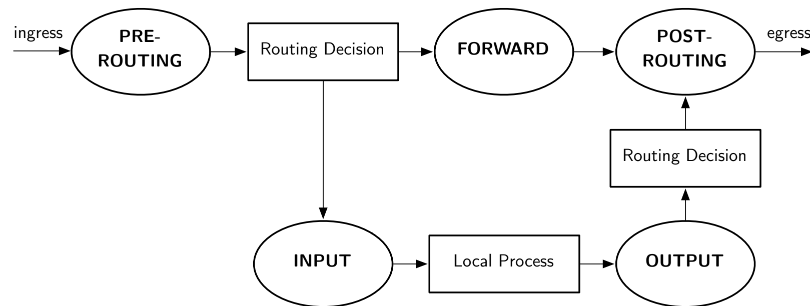


Figure 2.1 Different hooks provided by Netfilter.

- **OUTPUT:** This is the first hook that is traversed by outgoing packets. All packets that leave the local machine pass this hook.
- **POSTROUTING:** All packets that leave the local machine traverse this hook. It is called after any routing decision. The POSTROUTING hook is typically used to implement Source Network Address Translation (SNAT).

Figure 2.1 illustrates the Netfilter's hook-system. Netfilter provides an API for registering and unregistering a callback function to a given hook. Such callback functions typically return a value, the verdict that controls how Netfilter should further proceed with the packet. The following verdicts are currently defined in Netfilter:

- **ACCEPT:** The packet should also traverse any further hook.
- **DROP:** The packet should be silently discarded.
- **QUEUE:** The packet is passed to a userspace program, which will handle the packet.
- **REPEAT:** This verdict forces the packet to traverse the same hook again.
- **STOLEN:** The packet is silently held until something happens. This verdict enables that packets can be collected for further processing. This is used for dealing with fragmented IP packets.

Kernel modules such as *ip_tables*, *arp_tables*, and *ebtables* use these hooks to provide a more convenient way for defining rules for filtering and transforming packets. A well-known userspace tool for inserting such rules is *iptables*, which we also extend for loading and unloading the Lua scripts.

Having a closer look at the Netfilter internals, we can see that Netfilter itself does not provide a lot of functionality. It rather offers a framework, where several Linux Kernel Modules (LKM) register their services. In fact, packet matching as well as packet processing (as used for NAT) functionalities are implemented in several LKMs being loaded when needed. In order to simplify the development and integration of such LKMs, the project *Xtables-addons* [16] was set up. Using Xtables-addons, there is no need to patch or recompile the kernel. Furthermore, this framework can be used to easily install Netfilter extensions that are not yet accepted in the main kernel/iptables packages. PacketScript was hence developed using the Xtables-addons. Typically, Netfilter extension development involves the implementation of a LKM containing all important packet processing functionalities, as well as a userspace plugin needed by iptables when a new rule is injected. The following subsections describe these two software parts.

2.1.1 Userspace Plugin

The word “plugin” implies that there is a userspace application to be extended. In the case of PacketScript, the userspace tool *iptables* is extended by the PacketScript userspace plugin. This is necessary in order for iptables to load the corresponding LKM into memory as well as for knowing the proper format to copy the provided parameters from userspace to the LKM. Additionally, the plugin may provide functionality for validating the parameters and for presenting information on the command line about the usage of the module. Xtables-addons also simplifies the development of the userspace plugin.

2.1.2 Linux Kernel Module

Once a rule is validated by the userspace plugin, the data is copied from userspace to the LKM. The LKM provides a *checkentry* function being invoked whenever data has been copied to the LKM. Usually, this function does additional validation and some initialization work. For calling the checkentry function the LKM must already be loaded. Loading and unloading is normally done by some userspace tools, such as for example *modprobe*, *rmmod*; or in the Netfilter case, *iptables* may automatically load the LKM. The LKM provides a *module_ init* function, which is automatically called when the LKM is loaded. Additionally, it provides the *module_ exit* function, being called when the LKM is unloaded. These functions are typically used to initialize and shutdown the LKM. For instance a Netfilter extension (un)registers several callback functions within these functions:

- The *checkentry* function is called when a new rule copies some data from userspace to the LKM. This function validates the passed data, and may initialize some data needed within the target/match function.
- The *destroy* function is called when a rule is deleted. This function is typically used for freeing the resources allocated within the checkentry function.
- The *target/match* function is called when a packet is passed to the match/target extension. This function is used to process the packet in order to change its content or just for deciding whether the packet gets either accepted or dropped.

There are a few other functions, which are rarely used and remain unused within PacketScript. Further information about developing Netfilter extensions can be found in [17, 18].

2.2 Lua

Lua is an imperative scripting language released under the MIT license (since version 5.1). It comes with a lightweight script interpreter written in ANSI C that can be easily embedded in every C program. This enables that an application can be partially programmed in Lua. The Lua scripts used can be modified without recompiling the whole application, thus enabling a rapid development process. A very simple use case is the configuration of a program by a Lua script. More sophisticated is the ability to write a whole part of an application in Lua. Several professional applications, such as Adobe Lightroom or Blizzard's World of Warcraft, are partly developed in Lua [19]. There are several reasons for the success of Lua: On one hand it has several features C does not lend itself to: a good hardware abstraction, dynamic structures, no redundancies, ease of testing and debugging; on the other hand Lua comes with a safe execution environment, garbage collection, and facilities for handling strings and other data types with dynamic size.

The inventors of Lua decided to provide very expressive language constructs instead of blowing up the language with unnecessary language features, thus reducing the size of the language, its interpreter, and its API. Thanks to the expressiveness of the Lua syntax and the power of some language constructs, several programming paradigms are possible. Lua can be seen as a "multi-paradigm" programming language that enables common procedural programming, object oriented programming, and functional programming [20]. Since Lua does not target one single programming paradigm, it has no explicit support for object

orientation and inheritance, but with its metatables mechanism it easily enables their implementation. The same applies for namespaces and classes. Lua is a dynamically typed programming language, supporting only a few atomic data structures such as boolean values, numbers (double-precision floating point by default), and strings. Common data structures such as arrays, sets, lists, and records are represented with a Lua table. The Lua table is actually the single native data structure, which is basically a heterogeneous associative array [2].

2.2.1 Metatables

As mentioned before, Lua has some very expressive language constructs. One of these constructs are metatables and its corresponding metamethods. Such metatables provide some "type"-features to Lua tables, typically only available for numbers and strings. Such features are arithmetic and relational operators, but also concatenation as well as methods to obtain the size and string representation of a variable. Using metatables it is possible to define these operators also for Lua tables. It is quite useful to have such operators for tables, but they are mainly syntactical sugar to simplify the development. Besides these "common" operators, Lua provides a way to influence the normal behavior of a table during the query and modification of absent fields. This enables for instance, to write a metamethod that automatically creates a new value if an absent table field was queried. Or the function may lookup another table for the query. This is one of the basic building blocks for implementing inheritance. More details about Lua metatables can be found in *Programming in Lua* [2].

2.2.2 C API

Lua is designed to be embedded in C or C++ applications, but it was also used to extend software developed in Java, C#, Smalltalk, Fortran, Ada and Erlang. Lua offers the developer a rich API enabling a strong integration with code written in other languages. Lua manages a global stack being used for transferring parameters between Lua and C functions. As a result, Lua provides simple functions to push and pop most common C data types. Furthermore, calling Lua functions from C is also done using the stack. In this case the name of the Lua function and all its arguments are pushed onto the stack. A call to the proper C API function, for instance `lua_call` will then invoke this function with the given arguments. The Figure 2.2 depicts such a scenario, where the Lua function *max* is called with two integer values as arguments.

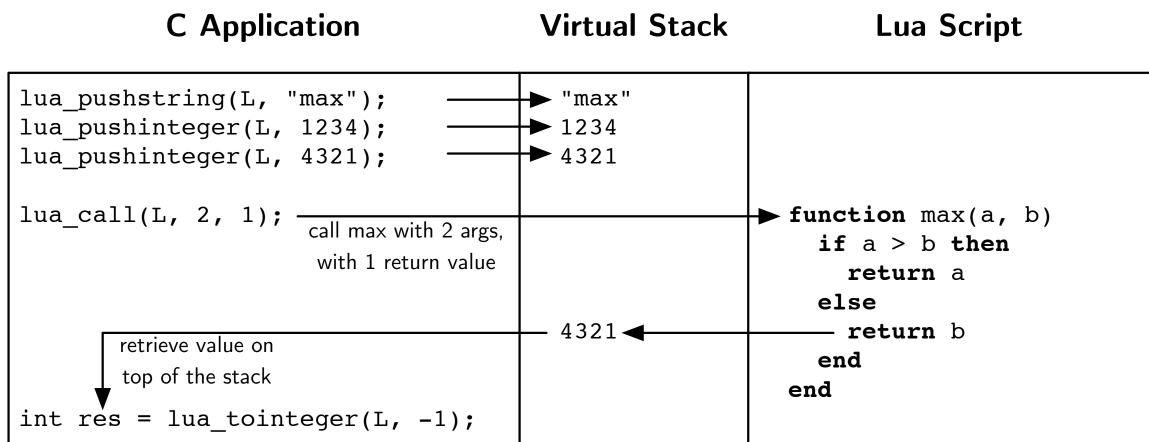


Figure 2.2 Calling a Lua function from C with two arguments

Chapter 3

PacketScript

This chapter describes PacketScript, the main contribution of this thesis. PacketScript is a scripting framework that changes the way of programming packet matching and processing logic for Linux Netfilter. It enables the development of Netfilter extensions entirely in Lua. In Section 3.1 we provide the specification of PacketScript. Based on this specification we built the architecture described in Section 3.2. In Section 3.3 we explain our concept of object oriented packet scripting and how it is applied within PacketScript. Sections 3.4 and 3.5 bridges the gap between the concepts and their application by providing some showcase examples. Sections 3.6 and 3.7 introduce two Lua libraries for dealing with bits and bytes, deferring work using the Linux work queue interface, and sending network packets using Lua.

3.1 Specification

This section specifies PacketScript in terms of functionality, interfaces, and performance. Moreover, it describes its nonfunctional requirements and the design constraints. The development of such a specification is typically an iterative process, as the software development is. Therefore, the specification may change with a next version of PacketScript. The keywords *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this specification are to be interpreted as described in RFC 2119 [21].

3.1.1 Functionality

The system shall intercept network packets within the Linux kernel. A Lua script (script) must be loaded and interpreted by the system. Such a script is then called to analyze and modify the intercepted packet. Typically, packet interception is done inside an interrupt handler. Spending too much time handling interrupts is not recommended, therefore the system should allow to defer work to a context that is better suited for processing parts of the script. The system must provide a convenient way for analyzing and modifying network packets for binary as well as for plain-text protocols. In order to use the system for prototyping network-centric applications in Lua, the system shall provide the functionality to send new network packets, obtaining accurate time stamps and random values.

3.1.2 External Interfaces

The system must be configured by a userspace application controlled by a privileged user (configuration interface). System configuration includes loading and unloading scripts, which may perform additional configuration. The configuration interface must be simple, therefore the file system path to the script should be the only parameter that is required. Besides the configuration interface, the script provides a more sophisticated interface to the user (script interface). It is used to control how a packet is analyzed, modified as well as how work is deferred. Therefore, such a script needs to be validated. The script interface is constrained by the syntax and semantics of the Lua scripting language. Additionally, the system must interface with the operating system using the Linux kernel API (operating system interface), for example for sending packets, deferring work, or obtaining accurate time stamps. On one hand, the system itself employs some functionality provided by the operating system interface. On the other hand, the script may call functions that wrap some of the operating system functionality. While function calls invoked by the system are hidden from the user, the functions called from the script are not. Since a wrong use of such functions can negatively affect the operating system behavior, they must be especially protected.

3.1.3 Performance

Intercepting network packets always introduces an overhead, which often leads to some performance penalty. However, the system itself should not produce a bigger overhead

than other packet intercepting solutions on uniprocessor (UP) systems or on symmetric multiprocessing (SMP) architectures. The system should not tackle a performance loss generated by complex or erroneous Lua scripts. Since the main use case of the system is prototyping new packet analyzers and modifiers, other performance guarantees are not required.

3.1.4 Nonfunctional Requirements

The system should be based on a modular architecture that is able to deal with evolving network protocols as well as new protocols. For this reason, the system must be easily extendable in order to support new network protocols. Such protocol extensions are statically compiled extensions written in C or programmed in Lua. All these requirements should enable the development of packet analyzers and modifiers in an easier and faster way using Lua.

3.1.5 Design Constraints

The system targets the Linux based operating system (Linux, kernel version 2.6.31) in collaboration with the Netfilter framework (Netfilter) and the Xtables-addons package. While Linux is available for several processor architectures, our system should focus on common x86 architectures. However, it is recommended to build an architecture independent system. Building the system should not involve the re-compilation of the Linux kernel or integral parts of it. The system should closely collaborate with Netfilter and extend its functionality where needed. Furthermore, existing userspace applications (e.g. *iptables*) used to interact with Netfilter should be extended for providing the configuration interface of the system. Netfilter provides hook handling in the kernel for intercepting and manipulating network packets. Since packet mangling typically takes place in the network and transport layer, the system should also be in line with these layers. For this reason, the system must support the Netfilter hooks available on network layer, but it may also support lower layered hooks.

3.2 Architecture

This section describes the architecture of PacketScript. As described earlier, Netfilter is extended by common Linux kernel modules, which are loadable by *insmod* or *modprobe*. Since Netfilter is extendable, it enforces a couple of architectural design decisions, such as how to

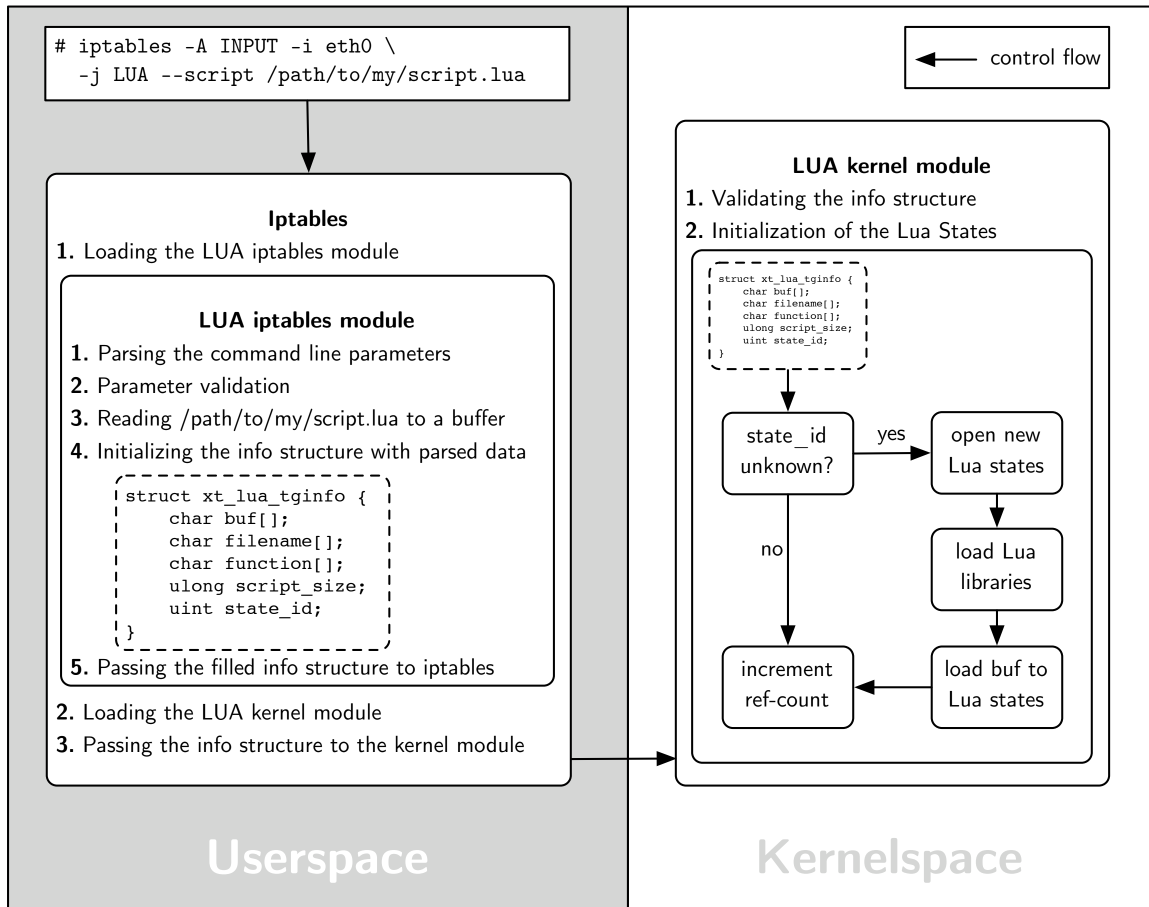


Figure 3.1 Illustration of the process when a new PacketScript rule is injected using iptables.

pass data from the userspace to the kernel module or how to register the different Netfilter callback functions. In addition, typical Netfilter extension development involves the realization of an iptables userspace plugin used for parsing and validating iptables parameters. Figure 3.1 illustrates the insertion procedure of a PacketScript rule using iptables. Since Figure 3.1 gives only an overview of the PacketScript's initialization process, the rest of this section is a detailed description of the iptables plugin and the Netfilter extension we developed. Moreover, it provides some insights about how the current version of PacketScript deals with concurrency.

3.2.1 Iptables Userspace Plugin

The iptables userspace plugin extends iptables in order to control which script is loaded or unloaded. We wanted to have a simple interface to PacketScript by favoring convention over configuration. The interface is as simple as having one mandatory iptables parameter `script` that takes the filesystem path of the Lua script. The plugin reads the script into a buffer and passes it together with some optional information to the Netfilter extension. The following paragraphs describe the conventions used and the possible options.

process_packet Function

The Lua script must contain a global function named `process_packet` taking at least one parameter, the packet. This function is called each time a packet matches the filtering rule and must return one of the Netfilter verdicts. Nevertheless, there are situations where several rules should use the same Lua script. For this reason, an optional iptables parameter, `function`, that takes the name of the callback function can be used.

Default Lua State

A Lua state can be seen as the execution context of the Lua virtual machine. For this reason, different Lua states are completely independent of each other. They share no data at all. PacketScript loads the Lua script into the *default* state. This results in all rules sharing the same Lua state, being useful in some scenarios but also dangerous in others. A typical scenario where a complete separation is of interest is when exactly the same script must be loaded several times. Such a situation may use the optional iptables parameter `state`, taking a positive integer identifying the Lua state. Without that separation, the script has to deal with the different access requirements for the global variables. As long as there are no global variables that are modified by different rules the optional `function` parameter will as well solve the problem.

3.2.2 Netfilter Extension

When a Linux kernel module is loaded or unloaded, their `module_init` or `module_exit` functions are called. As stated in Subsection 2.1.2, the Netfilter framework simplifies the development by providing hooks where extensions must register their functions. This is necessary in order to know which function to call when a new rule is loaded, when a rule gets

unloaded, or when a packet arrives. These functions are registered within `module_init` and unregistered in `module_exit`. PacketScript handles these callbacks as follows:

- **Rule Insertion:** PacketScript checks for an initialized Lua state according to the given state identifier. If the state is not yet initialized it loads the script and the required Lua libraries into the state. The same procedure is repeated once, resulting in an identical copy of the state that is used during software interrupt (softirq) handling. Since more than one rule can use the same Lua state a reference counter is incremented. In Section 3.2.3 we explain in-depth why an extra Lua state for handling softirqs is necessary.

- **Rule Deletion:** Depending on the reference counter of the Lua state, PacketScript closes the Lua state and frees all allocated resources. If a rule that uses the same Lua state still exists, only its reference counter gets decremented.

- **Packet Arrival:** Netfilter calls the registered function in the softirq context. Besides a reference to the packet, Netfilter also passes the information about what Lua function in which Lua state has to be called. Further, the function initializes a packet wrapper interpretable by Lua and calls the Lua function in protected mode. After the matching/processing work is done, the Lua function must push a Netfilter verdict onto the stack that is returned to Netfilter. Since the Lua call is protected, any error during the call is catched by the Lua-internal exception handling. In case of an error PacketScript drops the packet.

3.2.3 Software Interrupt Context vs. Process Context

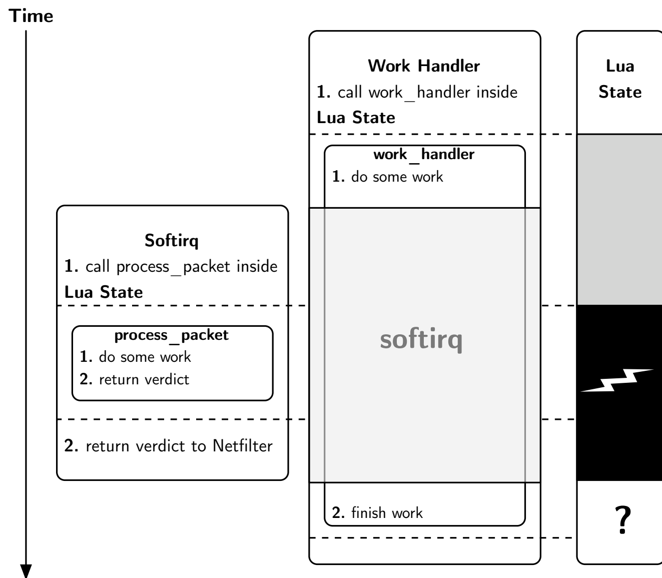
Whenever a network packet arrives, Netfilter calls the target/match function in the softirq context. Softirqs may run concurrently on all available CPUs. Typically, doing time-consuming work inside a target/match function decreases the network throughput. For this reason, it is possible to defer some work to the process context where function calls are even allowed to sleep. In order to defer such work, Linux provides the work queue interface. In PacketScript, we have developed a small Lua library providing an interface to its own work queue (see Subsection 3.7.1). However, this solution has a major drawback: if a softirq interrupts the kernel thread processing the deferred work, this can lead to a corrupted Lua state. PacketScript needs to deal with different race conditions on SMP architectures and on UP systems. Figure 3.2 depicts this problem for UP and SMP systems. Although not

described by this figure is the situation where two softirqs concurrently manipulate the Lua state (without a competing worker thread), which ends up in a race condition. There are two solutions to solve this problem. On one hand, the realization of proper locking inside the Lua VM by implementing the `LuaLock` and `LuaUnlock` functions [22]. On the other hand, the use of two distinct Lua states where one is used during softirq handling, and the other while processing the deferred work. The realization of the first approach is tricky, since PacketScript has different locking requirements:

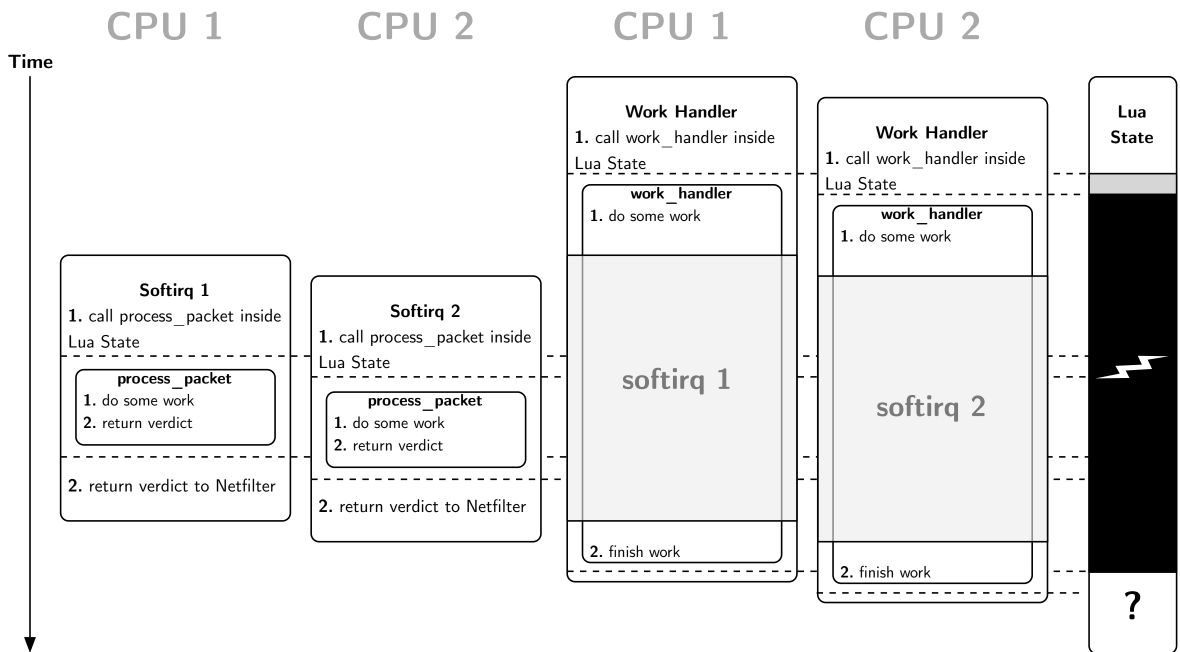
- **Target/Match Function:** On SMP architectures the target/match functions are concurrently called back in softirq context. Inside the callback function the access to the Lua state requires to be synchronized. The usual protection for such cases is `spin_lock_bh`. It implements the standard spinlock mechanism and it disables softirqs. The use of such a spinlock is a kind of busy waiting. Depending on the script, the Lua function call inside the target/match function triggers several calls to the Lua VM. As a result, the `LuaLock` and `LuaUnlock` functions are called several times. Since the Lua call itself is protected by `LuaLock/LuaUnlock` we will end up with recursive locking, which is not possible using `spin_lock_bh`.

  Recursive Locking: A thread may acquire the same lock recursively without having to unlock it first. In Linux spinlocks, the locking tool of choice inside interrupt handlers, are not recursive and a re-acquire will end in a deadlock. Nevertheless, one exception exists: the so called "Big Kernel Lock" (BKL), which is a recursive spinlock. However, using the BKL is highly discouraged and a lot of discussions about removing the BKL from the Linux kernel are going on.

- **Deferred Work:** Inside a kernel thread the linux work queue executes the deferred work in process context. A softirq may interrupt a still running worker thread, yielding a corrupted Lua state. The usage of `LuaLock/LuaUnlock` will likewise end in a recursive locking scenario. This problem is even harder to solve since additional race conditions owing to the concurrent workers exist. Even though, in process context such race conditions may be avoided by using a semaphore acting as a mutex. However, inside the softirq context the locking problem remains unsolved.



(a) Race condition on UP system



(b) Race condition on SMP system

Figure 3.2 In Figure 3.2a a typical race condition existing on an UP system is depicted. While handling a softirq, an existing worker thread may be interrupted, leaving the internal state of the Lua VM in an unpredictable state. The scenario on a SMP system depicted in 3.2b is even worse: While the concurrently handled softirqs already introduce the need for locking, the interruption of the concurrent worker threads (themselves competing in a race) screams for additional synchronization too.

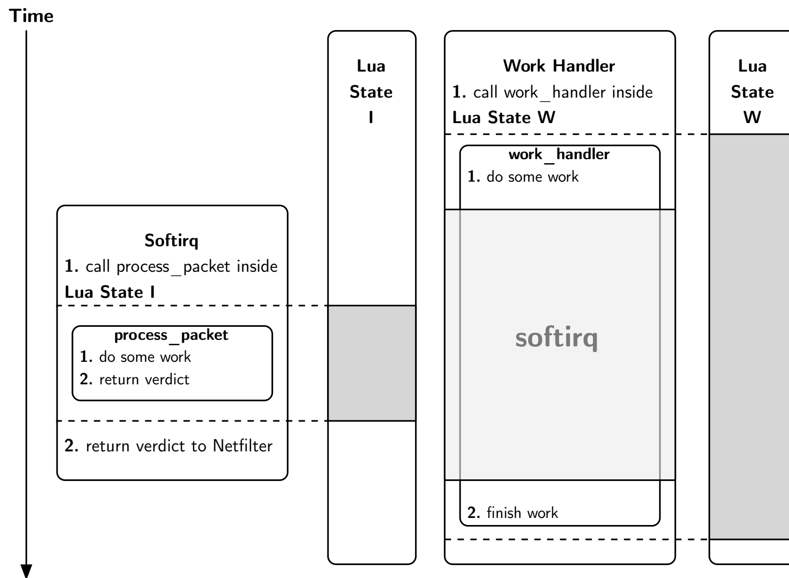
Possible Solution: The "Big Kernel Lock" may work, but we did not test it. A possible solution, based on `LuaLock/LuaUnlock`, would possibly imply a modification of the Lua VM in order to support ownership-based locking, and coming up with an own recursive, owner-aware locking solution that can be applied in either softirq or process context.

Even if the development and integration of such a locking solution is highly interesting, we decided for the sake of simplicity to implement two distinct Lua states that are loaded with the same Lua script. Unfortunately, this solution makes the development harder for the programmer because he has to deal with two implicit programming scopes. Both scopes are loaded with the identical global environment, but since the two Lua states do not share any global variable, it is up to the programmer to know which variable was changed in what way. In order to assist the developer with scoping problems, PacketScript provides the constant `IS_INTERRUPT`, defined inside the Lua state handling the softirq.

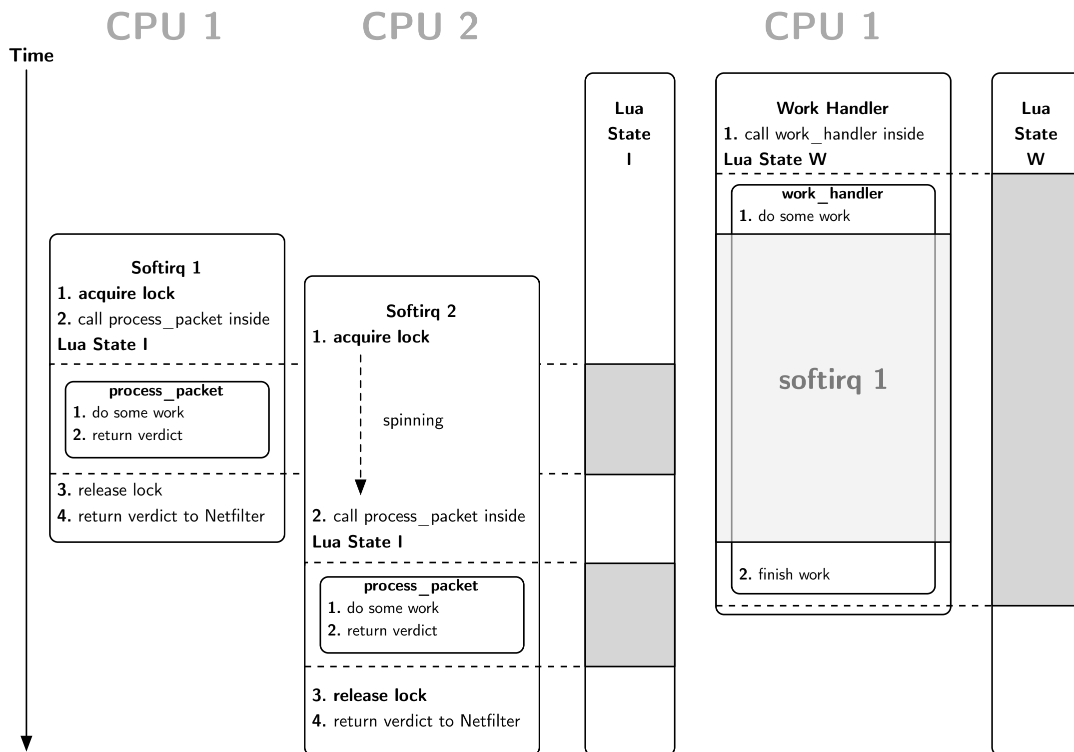
As already mentioned, softirqs run concurrently on SMP. For this reason, PacketScript protects the Lua function call by acquiring a spinlock. Furthermore, the linux work queues are processed by default by as many threads as the number of internal CPUs. Therefore, the work queue is configured to use only one single worker thread, which reduces the locking overhead. Figure 3.3 depicts how PacketScript avoids race conditions.

Despite the mentioned drawbacks using two distinct Lua states, our solution is simple. Besides its simplicity there is another key benefit: memory management. Processes running in kernelspace typically use a "may-sleep" allocator. The allocator can put the process to sleep, waiting for a free page when invoked in low-memory situations. While the process is sleeping, the kernel locates some free memory, either by flushing buffers to disk or by swapping out memory from user processes. However, this is not possible when the allocator is invoked from outside the process context, as it happens when handling a softirq. In such situations an atomic memory allocator must be used, which can even take the last free memory page [23]. Two distinct Lua states easily enable different Lua allocator functions. As a result, PacketScript uses atomic allocation while handling softirqs and a normal "may-sleep" allocation when deferred work is processed.

However, it is not optimal that in the age of symmetric multiprocessors, PacketScript is still on the sequential trail. We hope that we could tackle this problem in a next version of our implementation.



(a) Solved race condition on UP system



(b) Solved race condition on SMP system

Figure 3.3 Since two distinct Lua states are involved, one used for handling the softirq the other used while work queue processing, no race condition exists anymore on UP systems depicted in Figure 3.3a. In order to bypass the race conditions on SMP systems, a spinlock is used while concurrently handling the softirqs. Furthermore, the work queue is limited to a single worker thread as illustrated in Figure 3.3b.

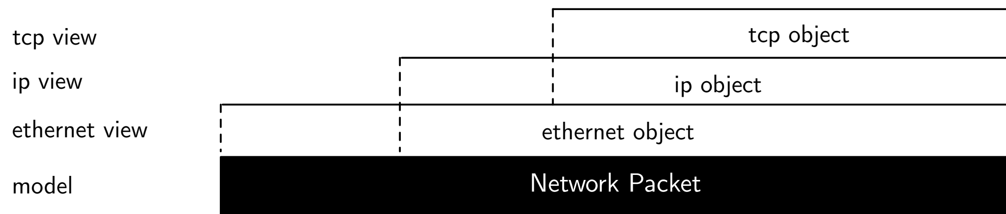


Figure 3.4 Different views on an Ethernet frame

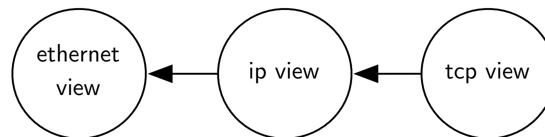


Figure 3.5 Dependency graph

3.3 Object Oriented Packet Scripting

One objective when designing PacketScript was to access the fields of a packet header in an object oriented style using Lua. Since PacketScript should be extendable and not being restricted to a small set of networking protocols, it was important to design an easy to use generic framework enabling object oriented packet scripting. To be more specific:

- A network header of protocol X is an object of class X.
- Class X contains the methods to dissect each field of protocol X.

Normally, network packets are structured by several nested protocols. For example, an Ethernet frame contains an IP packet, which itself encloses a TCP segment. For dissecting each header within such a network packet, three objects must be created. Each one provides a specific view of the underlying memory area. Figure 3.4 illustrates the protocol nesting and the views provided by the objects. Such a protocol nesting introduces some dependencies on the views:

- A view defines where a sub-view may be applied.
- A view defines if a sub-view may be applied.

Given these dependencies, a view cannot be created without further knowledge of its parent view. The simple dependency graph for our Ethernet example is shown in Figure 3.5.

Since the nesting of protocols is common in networking, PacketScript implements a factory approach, where a view is created by its parent view. As a result, our specification of object oriented packet scripting must be extended:

- One field of protocol X may contain payload data of another protocol Y.
- The class of protocol X provides a factory method producing an object of class Y.

However, there are other approaches, which are not considered by PacketScript, to bypass these dependency problems, for example the use of global knowledge or constraint satisfaction problem solvers.

3.3.1 Generation of Protocol Classes

In PacketScript, the generation of Lua classes as well as their instantiation is usually performed by the framework. While the classes are dynamically created when PacketScript is loaded into the Lua state, their instantiation is implicit. As quickly mentioned in Subsection 3.2.2, the Netfilter callback function initializes a packet wrapper interpretable by Lua, which is an object of type *raw* (hereinafter referred to as "*raw* object"). The view provided by such a *raw* object is protocol independent. As a result, any other protocol class can be instantiated. Such a generic factory is favorable for two reasons: First, the network packet may still be incomplete (e.g. Ethernet header is missing) depending on the processing Netfilter hook, typically the output and postrouting hook. Second, PacketScript does not rely on the link layer protocol used. As a consequence, the script developer must be aware of the underlying packet structure. Further development of PacketScript may improve this by rather passing an instance of the proper class instead of using such a generic approach.

The current version of PacketScript will pass a *raw* object, which provides the overall view. That means a packet may start with the Ethernet header, while the IP header is of interest. Consequently, this requires one implicit and two explicit class instantiations: The *raw* object is implicitly created and passed to the Lua function. Within this function an explicit call to the factory method of the *raw* object returns a new instance of the class *ethernet*. Finally, this new object serves as a factory to generate an object of type *ip*. Figure 3.6 illustrates an identical scenario. So far we covered the generation of the classes, but not the generation of the class dissector methods. Their generation is a bit more complex, since our framework must be able to deal with binary as well as plain-text protocols. Typically, binary protocols consist of fields of a given length, placed at some

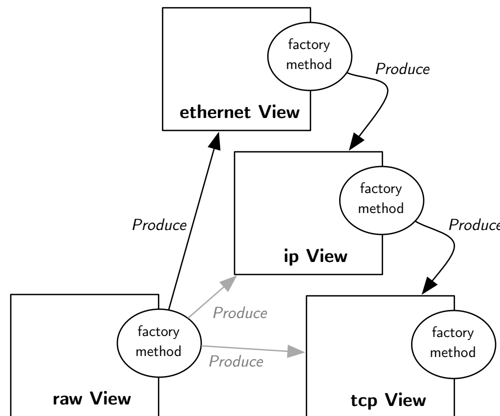


Figure 3.6 Codechart describing the extraction of the TCP header within an Ethernet frame, in LePUS3 notation [24].

offset within a chunk of bytes. Such information is usually well described in the Request for Comments (RFC) published by the Internet Engineering Task Force (IETF) for the protocol. However, for plain-text protocols these lengths and offsets are normally unknown and must be calculated by parsing the network header. Additionally, the order of the fields is usually unknown too, which complicates the process of calculating the proper lengths and offsets. In either case the field lengths and offsets are essential. For this reason, we think it is enough to store the offset and lengths of a field in order to generate its dissector function.

In order to generate such classes, they must have

- a unique name, resulting in the class name;
- a list of protocol fields, each containing a name, offset, and length; and
- a function, responsible for controlling its factory behavior.

In PacketScript all this information is stored within a protocol specific C structure, or even more dynamic, inside a table within the Lua script. Such a structure is passed to a class generator, heavily using Lua's metatable mechanism. The following enumeration describes the class generation process:

1. PacketScript creates a new metatable using the unique class name.
2. For each protocol field a new variable is created within this metatable, which takes the field name.

3. A generic field function is assigned to the variable. PacketScript passes a reference to the field as upvalue[1],such that the function is able to determine the proper offset and length, as well as to control the proper injection[2] of the field modifier function.

Such a generation of classes has one major drawback for object oriented packet scripting: Without directly manipulating the list of protocol fields there is no way to change the field offset and length later. But, this is quite common for more sophisticated protocols. For instance, the size of the TCP options field is constrained by the data offset field inside the TCP header, resulting in a change of the options length and in a shift of the data offset. Therefore, a protocol may provide an optional function, intercepting the calculation of field offsets and lengths for each packet. This enables a dynamic adaptation of protocol fields.

Following this design, we are able to perfectly dissect a protocol header in an object oriented fashion.

3.3.2 Field Modifiers

The previous subsection presented the requirements for creating the protocol classes. But, we did not cover how the objects are used to modify the content of a network packet. PacketScript injects the code for retrieving and manipulating the values. Such logic must be as generic as possible in order to deal with all the different types of field values.

The most basic way of accessing fields follows a byte-oriented approach, where each byte of a field is separately addressed. Although such a general solution is a nice start, there are cases where a more sophisticated approach is needed. For one thing, Lua has no binary operators, which makes it hard to segment less than one byte. Thus, PacketScript provides a way to control what field access and manipulation functionality it is going to inject. Within the list of protocol fields, each field may provide references to a *getter* and *setter* function that are injected, enabling a more specific access instead of the very generic byte-oriented approach. For example, this is used to get/set integer values, strings (e.g. IP addresses or MAC addresses).

---

[1] Due to lexical scoping, local variables can be freely accessed by functions defined inside their scope. A local variable used within an inner function is called an *upvalue*, or external local variable, inside the inner function [2].

[2] Dependency injection is the concept of providing an external dependency to an application. Generally spoken, indicating to an application what other logic it can use. This concept is well known in object oriented programming. However, programming Lua scripts, which are loaded at runtime, can be seen as a special type of dependency injection.

PacketScript encapsulates these optional functions as well as the structure holding the information presented in 3.3.1 in small Lua libraries. Within PacketScript these libraries are called *Protocol Buffers*[3]

3.4 Protocol Buffers

The section before introduced the concept of object oriented packet scripting as well as it provided some details how the concept was realized within PacketScript. However, so far we have not seen any working code. This section demonstrates how the protocol buffers are applied in order to dissect a network packet as well as its fields are accessed in various different ways.

3.4.1 The Raw Approach

As roughly explained, the root of each object is the *raw* object, which can be used to dissect a packet and change the values of the fields. Although the available functions are very generic, they provide everything which is used to access the content of a packet in a byte-oriented way. The Listing 3.1 shows how to dissect an Ethernet header.

Listing 3.1 Segmenting the fields of the Ethernet header using `raw`

```
1 function process_packet(p)
2     dmac = p:raw(0,6)
3     smac = p:raw(6,6)
4     type = p:raw(12,2)
5     ...
6 end
```

Whenever a packet arrives, the Netfilter extension calls the function `process_packet`, passing a new *raw* object to the function. Since Lua is a dynamically typed programming language every Lua type could be passed to this function. But, as a convention the Netfilter extension passes a *raw* object. The second line shows how to manually segment the field containing the destination MAC address by calling the `raw` function with a given byte offset and length. This results in a new *raw* object, which could be further segmented. The same is repeated for the field containing the source address and the type.

[3] The term "protocol buffer" appeared in the *Google Developer Guide* [25], but the usage is different. They are using protocol buffers to serialize data structures in order to use them between different programming languages. Although the mechanisms are different, we share an identical intention.

Once the dissection is done, we may care about retrieving the values of the segmented fields. Listing 3.2 shows how to retrieve the value of the destination MAC address.

Listing 3.2 Retrieving the value of a segmented field as a byte array using `get`

```
1  function process_packet(p)
2      ...
3      val = dmac:get()
4      byte1 = val[0]
5      byte2 = val[1]
6      byte3 = val[2]
7      byte4 = val[3]
8      byte5 = val[4]
9      byte6 = val[5]
10     ..
11 end
```

The object `dmac` is of type *raw*, meaning that it can be further segmented if needed. Besides this functionality it also provides a function for retrieving its value. A call to the function `get` creates a byte array holding the data that can be interpreted by Lua. The lines 4 - 9 show how to access the different bytes within this byte array. Of course it is also possible to set the values of such an array by calling e.g. `val[0] = 0xFF`. It is important to mention that PacketScript does not copy the bytes to the Lua state when calling `get`. It manages a pointer to the start of the memory area. As a result, the developer must be aware of the underlying memory management processes, as a C pointer can be freed without notifying the Lua garbage collector. This is typically the case when operating on a network packet, whereas the memory of the underlying *sk_ buff* structure is freed by the Linux kernel. Normally, we do not have to care about this issue. But, whenever the same byte array should also be used within a further callback or inside a deferred work we must consider to explicitly copy the bytes to the Lua state. Another way of modifying the actual packet content in *raw*-mode is using a memset-alike approach. The Listing 3.3 demonstrates how to set the values of a segmented field. A call to `set` will set every byte inside this segment to the given byte value.

Listing 3.3 Setting of a range of bytes using `set`

```
1 function process_packet(p)
2     ...
3     dmac:set(0xFF)
4     ..
5 end
```

In order to set the single octets of the destination address seperately using `set`, each byte has to be segmented and manually modified. This is shown in Listing 3.4.

Listing 3.4 Setting a single byte using `set`

```
1  function process_packet(p)
2      ...
3      dmac:set(0xFF)
4      byte1 = dmac:raw(0,1)
5      byte2 = dmac:raw(1,1)
6      byte3 = dmac:raw(2,1)
7      byte4 = dmac:raw(3,1)
8      byte5 = dmac:raw(4,1)
9      byte6 = dmac:raw(5,1)
10
11     byte1:set(0xFF)
12     byte2:set(0x00)
13     ...
14 end
```

Obviously, the *raw*-approach has its value, it is the tool of choice when byte-oriented access is enough or the packet structure is simple. However, the last example demonstrated that already the manipulation of a simple Ethernet MAC address demands quite some amount of code. The next section shows how it is possible to set the Ethernet source address by simply calling `dmac:set("FF:FE:FD:FC:FB:FA")`.

3.4.2 Structured Packet Access

As seen before, directly operating on the packet content is quite tedious. Protocol buffers may help here. As described in Section 3.3, PacketScript may provide specific getter and setter functions enabling a more convenient way of changing the packet content. This section describes the Ethernet protocol buffer, enabling a direct comparison to the *raw* approach. The section before did not cover the special *raw*-ability of producing the objects of all the

different protocols. The *raw* factory method is named `data`. Listing 3.5 shows how to create a new *ethernet* object using the *raw* factory.

Listing 3.5 Generation of an *ethernet* object using the *raw* factory method `data`

```
function process_packet(p)
    eth = p:data(packet_eth)
end
```

The resulting *ethernet* object now provides all the defined *getter* and *setter* functions. This enables a more sophisticated way of dissecting and manipulating the fields. Listing 3.6 demonstrates a simple MAC address translation example using the *ethernet* protocol buffer.

Listing 3.6 Demonstration of the *ethernet* protocol buffer

```
function process_packet(p)
    eth = p:data(packet_eth)
    smac = eth:smac()
    dmac = eth:dmac()

    if smac:get() == "01:23:45:67:89:AB" then
        dmac:set("FF:FF:FF:FF:FF:FF")
    end
    ...
end
```

Compared to the *raw* approach, modifying the packet content is easy. There is neither the need for manually segmenting the MAC addresses nor the bytes must be dissected separately. In contrast, the *ethernet* object provides functions to dissect the fields, enabling an easy way for modifying their value. A call to the factory method of the *ethernet* object further dissects the payload data. Listing 3.7 shows how to extract the IP header and how to get its source and destination address using a shortcut.

Listing 3.7 Extracting the IP header from an Ethernet frame

```
function process_packet(p)
    eth = p:data(packet_eth)
    ip = eth:data(packet_ip)

    saddr = ip:saddr():get()
    daddr = ip:daddr():get()
    ...
end
```

Both calls to the factory methods of the *ethernet* and *raw* object look similar, but only the call to the *ethernet* object is validated. The factory method consults the type field within the Ethernet header for deciding if its payload contains an IP packet. The call to the factory function will not succeed if it does not contain an IP packet.

The Factory Function. In PacketScript a call to the factory function will trigger a "policy" function, which is defined in each protocol buffer. This policy function has full access to the packet, allowing it to inspect the content of the packet. The policy function just needs to return a boolean value that indicates if the object can be created. As a particular case, the policy function of the *raw* object does always return true.

The Pre-Calculation Hook. Besides the policy checking mentioned, the factory function may call a hook provided by the protocol buffer to manipulate the calculation of field offsets and lengths. Such a hook must return two vectors having at least as many elements as protocol fields. One vector is needed for the field lengths, the other for the offsets. While calculating the field offsets, the vector entry at index i is added to the field offset at index i. The same is done for the field lengths using the length vector. Interestingly a simple interception of calculating the field parameters is absolutely sufficient to deal with various different protocols. The TCP protocol buffer uses it for coping with changing parameters for its options and data fields. Such hooks may become quite complex as it can be seen in the source code of the HTTP protocol buffer. In the case of HTTP there is no previous knowledge available about the various field parameters. Therefore, the hook parses the HTTP header to correctly set the field offsets and lengths.

3.5 Dynamic Protocol Buffers

As quickly introduced in Subsection 3.3.1, PacketScript is able to create classes from either a specification written in C or in Lua. Whenever such a specification is provided by the Lua script we call it a *dynamic protocol buffer*. Listing 3.8 shows such a dynamic protocol buffer for the Ethernet protocol:

Listing 3.8 A dynamic protocol buffer for the Ethernet protocol header

```lua
function eth_policy(seg, type)
        return true
end

function eth_precalc(seg)
    local offset = {0,0,0,0,0}
    local length = {0,0,0,0,0}
    -- check if frame is vlan tagged according to IEEE 802.1Q
    if seg[12] == 0x81 and seg[13] == 0x00 then
        -- change offset of the type and data fields
        offset = {0,0,0,128,144}
        -- change length of the vlan field
        length = {0,0,32,0,0}
    end

    return offset,length
end

local eth_prot_buf = {
    name = "packet_eth_dyn",
    payload_field = "data",
    protocol_fields = {
        {"dmac",   0, 48, nil, nil},
        {"smac",  48, 48, nil, nil},
        {"vlan",  96,  0, nil, nil},
        {"type",  96, 16, nil, nil},
        {"data", 112,  0, nil, nil},
    },
    has_protocol = "eth_policy",
    get_field_changes = "eth_precalc"
}
register_dynamic_protbuf(eth_prot_buf)
```

The Lua table `eth_prot_buf` (Line 19) holds all information necessary for generating the class. This table is passed to the `register_dynamic_protbuf` function (Line 32), which prepares the data such that the class can be generated according to the classes specified in C. More specifically, it fills the same C structure. Since the hooks for the factory function and pre-calculation, as well as for *getter* and *setter* modifier functions are registered using common C function pointers we implemented generic wrapper functions. These wrappers

are able to call the provided Lua functions. The following paragraphs describe how such a dynamic protocol buffer must be specified.

The name of the class is indicated by the name of the protocol buffer (Line 20). The factory method of the class depends on the protocol field containing the payload. Therefore, the payload field is specified (Line 21). For generating the dissector functions a list of protocol field specification must be given (Line 22). Each specification is itself a Lua table, holding the field name, offset (in bits), length (in bits), and the optional *getter* and *setter* modifier functions. This implementation does not specify its own field modifiers and hence falls back to the *raw* approach. The policy function that is needed to control the factory behavior must also be given. Protocol buffers use the variable `has_protocol` for storing the name of the Lua policy function. Our example above implements this function `eth_policy`, which receives the whole header as well as the requested protocol type passed as arguments. Our implementation of the policy function does not care about the payload, therefore it always returns *true*. If the pre-calculation hook should be used, the name to such a pre-calculation function must be stored in the `get_field_changes` variable. In our implementation we use this hook in order to deal with VLAN tagged Ethernet frames. Whenever a frame is tagged the type and data fields must be shifted 4 bytes to the right. The VLAN tag information is then stored in the area between the source MAC address and the Ethernet type field. The pre-calculation function `eth_precalc` (Line 5) returns two Lua tables, one for changing the offsets and one for the lengths.

3.6 Byte Arrays in Lua

As mentioned in Subsection 3.4.1, the *raw* access is byte-oriented. To be more concrete, the byte arrays are objects like packets are. Although the most important functionality of such byte arrays is already covered, still a few interesting features are missing.

3.6.1 The *byte_ array* object

The values of a *byte_ array* object can be accessed and manipulated using the common array-notation. Besides this functionality we implemented the additional support for Lua's *to_ string*, *length* and *concat* operators.

3.6.2 The Bytes Library

We developed a library, which is used for dealing with such byte arrays. The library provides the following functionality:

- `bytes.new(`*`length`*`)`: The most common way to create a new *byte_ array* object.
- `bytes.new_from_string(`*`"hello world"`*`)` creates a new *byte_ array* object that is as long as the given string (including the terminal symbol).
- `bytes.many_to_one(`*`table_of_byte_array_obj`*`)` aggregates as many *byte_ array* objects stored inside the given table to one single *byte_ array* object, still preserving the order of the bytes. This function should only be used in process context, since it uses *vmalloc* for being able to allocate enough memory for the newly created object.
- `bytes.to_bytes(`*`packet_obj, protocol`*`)` converts a packet of any protocol to a *byte_ array* object.
- `bytes.to_packet(`*`byte_array_obj, protocol`*`)` converts a *byte_ array* object to a packet of the given protocol. Such a conversion avoids any factory policy. Nevertheless, the pre-calculation hook is called.

The last two conversion functions do not copy the data. Therefore, you must make sure that the underlying chunk of bytes, either hidden in a *packet* or *byte_ array* object, will not be destroyed by Lua's garbage collector.

3.7 The Netfilter Library

We also implemented the Netfilter library (`nf`), which contains a few functions that wrap some Linux specific functionality.

- `nf.get_random()` is a wrapper around a call to `get_random_bytes`. The random value is pushed as a 32 bit integer value.
- `nf.get_time()` is a wrapper around a call to `jiffies_to_msecs` in order to get an accurate timestamp in milliseconds.
- `nf.schedule(...)` wraps the logic for deferring work using the Linux work queue interface. We describe in-depth this functionality in Subsection 3.7.1.

- `nf.create_packet(...)` wraps the logic for sending new network packets from within a Lua script; Subsection 3.7.2 provides more details.

3.7.1 Deferring Work

As already mentioned, the softirq context is not appropriate for every kind of work (e.g. calling functions that may sleep). To handle that constraint, PacketScript provides the `nf.schedule` function in order to defer work to process context.

- `nf.schedule(nil,"callback_function",id,delay,[packet,protocol])` defers the execution of the Lua function named *callback_function*. An additional *id*, an integer value, and the *delay* in milliseconds must be provided. If *id* or *delay* are not needed, 0 can be passed. The first parameter, here *nil*, is used to pass a reference to the Linux *sk_buff* structure to the process context. Subsection 3.7.2 describes one reason for having such a possibility. The *id* parameter can be used when a single integer value is enough to control the execution of the callback. A typical use case would be a simple callback handler operating in process context that dispatches the callbacks according to the received *id*. When the optional parameters are provided, the *packet* object of type *protocol* is passed as parameter to the given callback function.

3.7.2 Sending Packets

As specified in Section 3.1, PacketScript should be able to send network packets. While we implemented this feature, it is currently limited to IP packets.

- `nf.create_packet(skb, byte_array)` creates a new network packet. This function wraps among others the calls to *ip_route_me_harder* and *ip_local_out*, which we use to send a packet. The parameter *skb* is needed to properly construct a new *sk_buff* structure. The provided *byte_array* object should contain a proper IP header in order for the function call to succeed.

Note that, the function for creating and sending new network packets is still experimental. There are surely better ways to send network packets from within a kernel module, even without keeping a reference to a *sk_buff* structure. In this case we took a pragmatic approach for the first prototype of PacketScript.

As stated at the very beginning of this chapter, the development of PacketScript was an iterative process. Typically, the iterations were used to refactor the software, to add new features, and to optimize the code. Since performance is the critical element of success in most network-centric applications, we spent quite some time optimizing PacketScript. The following chapter presents two experiments, which were set up in oder to compare the performance of PacketScript with hard-coded solutions provided by Linux. The results achieved provide a valuable feedback for further optimization rounds.

Chapter 4

Experiments

The previous chapter described PacketScript, which provides a novel way to develop Netfilter extensions, and this entirely in Lua. For testing our prototype and measuring the network performance, we developed a Netfilter NAT using PacketScript. The simple NAT implementation and its performance results are described in Section 4.1. As a further test scenario we built a solution for caching TFTP and HTTP traffic, combining various advanced features of PacketScript. Section 4.2 presents the scenario and the performance results of the cache. The results of both experiments are discussed in Section 4.3.

4.1 Network Address Translation

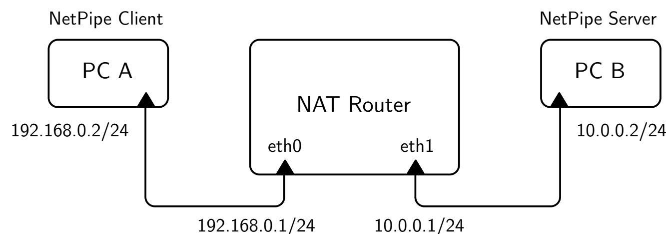


Figure 4.1 A simple NAT network

This experiment compares the performance of a common Linux NAT configuration with a NAT implemented using PacketScript. In order to show that PacketScript is able to deal with a high network load, we built a simple NAT Netfilter extension. The network depicted in Figure 4.1 was used throughout our test cases. We used two identical DELL Latitude

110L notebooks for PC A and B (Intel Celeron M CPU 1.3GHz, 512MB RAM). The NAT router is a DELL Optiplex 170L workstation (Intel Celeron CPU 2.4Ghz, 512MB RAM). All of them were running Ubuntu 9.10 Karmic Koala using the 2.6.31-21 kernel provided by Ubuntu. The network in both cases consisted of a dedicated 100Base-TX link between the PCs and the router. The router itself has a built-in onboard network card as well as a separate PCI network card. Additionally, the NAT router used iptables version 1.4.7. Listing 4.1 shows the Lua script that realizes our simple NAT solution.

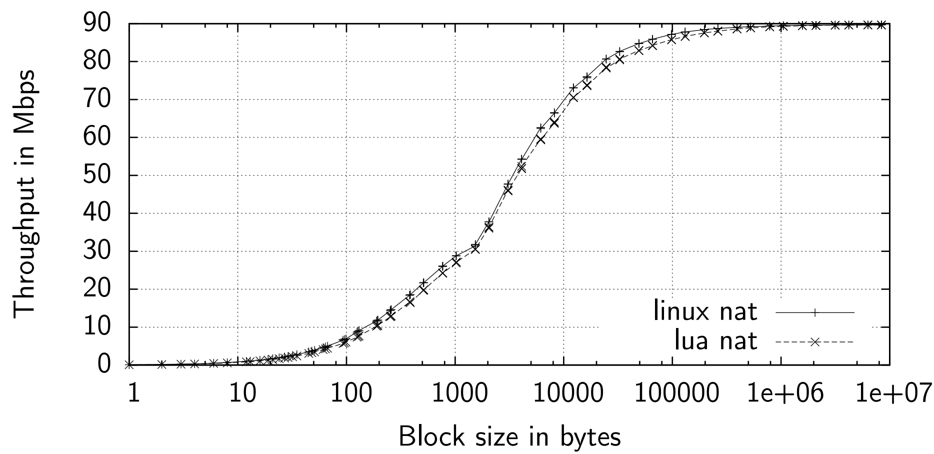
Listing 4.1 Simple NAT solution

```
1  -- Translating the source address for outgoing packets
2  function process_packet_snat(p)
3          local ip = p:data(packet_eth):data(packet_ip)
4          if not ip then return NF_DROP end
5
6          ip:saddr():set("10.0.0.1")
7
8          return XT_CONTINUE
9  end
10
11 -- Translating destination address for incomming packets
12 function process_packet_dnat(p)
13         local ip = p:data(packet_eth):data(packet_ip)
14         if not ip then return NF_DROP end
15
16         ip:daddr():set("192.168.0.2")
17
18         return XT_CONTINUE
19 end
```

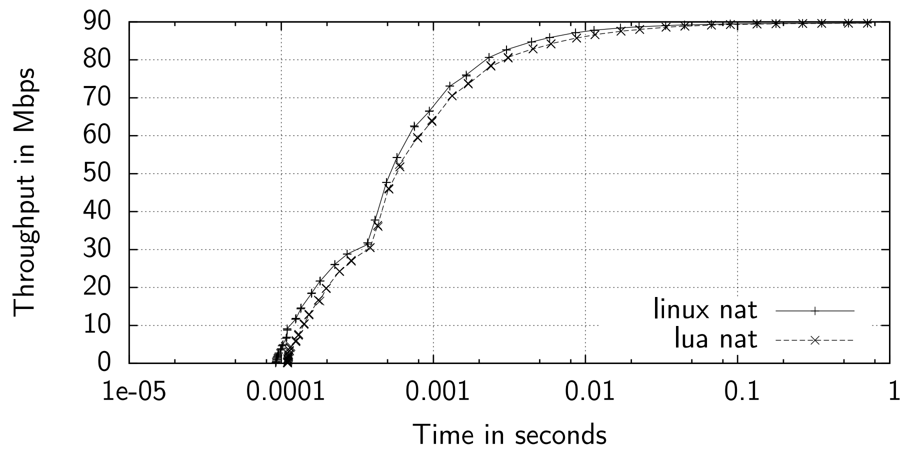
The following *iptables* rules are used for loading the script:

```
iptables -t mangle -A POSTROUTING -o eth1 \
        -j LUA --script nat.lua --function process_packet_snat
iptables -t mangle -A PREROUTING -i eth1 \
        -j LUA --script nat.lua --function process_packet_dnat
```

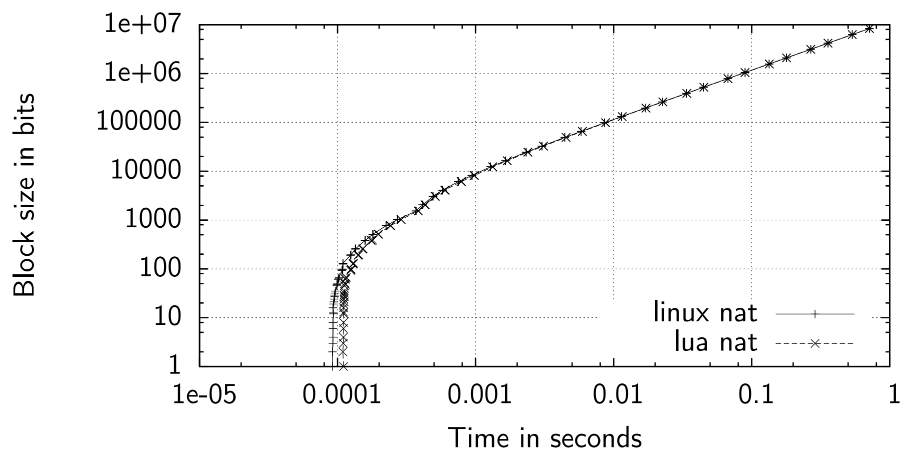
For generating load and measuring the performance we used NetPIPE [26], a network protocol independent performance evaluator. NetPIPE is based on the principles presented by the HINT [27] computer performance metric. HINT states that a computer's performance



(a) Throughput Graph



(b) Signature Graph



(c) Saturation Graph

Figure 4.2 Typical graphs resulting from a network benchmark using NetPIPE. Figure 4.2a compares the NAT throughput achieved using Linux SNAT/DNAT with the PacketScript NAT solution. Another important statistic is the network latency, shown as a NetPIPE signature graph in Figure 4.2b, again for both scenarios. Figure 4.2c shows the NetPIPE signature graph for both scenarios.

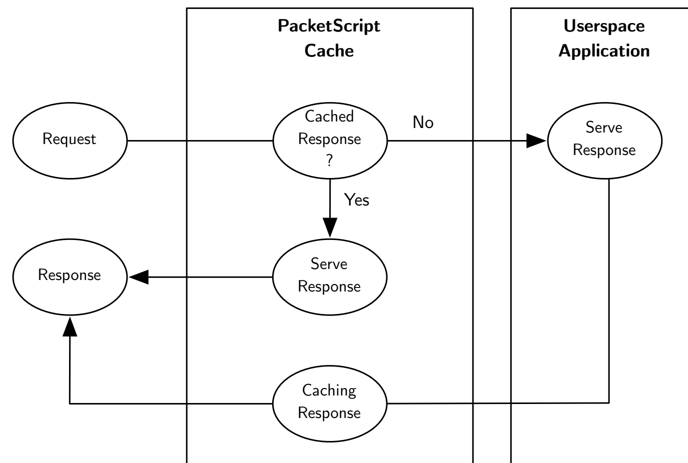


Figure 4.3 PacketScript checks an incoming request for its cached response. If a cached response exists for the given request, PacketScript directly serves the response. But, if there is no cached response for a given request, the request is passed to the userspace application and its response gets cached.

cannot be accurately described with a single sized computation. Similarly, the network performance cannot be described using a single sized communication transfer. NetPIPE increases the block size from one byte until the transmission time exceeds one second. Figure 4.2a presents the throughput versus the transfer block size for our test setup. Considering this Figure, the maximum throughput for either the Linux NAT or the PacketScript NAT is approximately 90 Mbps. However, another important statistic is the latency of a network, which is difficult to analyze using this graph. Therefore, NetPIPE is used to produce the *network signature graph*. Such a graph nicely visualizes the transfer speed versus the elapsed time, which represents the network latency. This latency graph for our test setup is plotted in Figure 4.2b. This graph shows that both setups have approximately the same latency. Another interesting statistic is the *saturation point*. This is the point after which an increase in block size results in a near-linear increase in transfer time. This statistic is shown in Figure 4.2c. This graph shows that both NATs have approximately the same saturation interval. Within this interval, both graphs monotonically increase at a similar constant rate. As a result, in neither scenario the network throughput can be improved by increasing the block size.

4.2 Application Level Packet Cache

The previous section compared the network performances of a NAT network using Linux boardtools with a NAT realized using PacketScript. Although these results look very promising, it is a relatively simple task using only a few features of PacketScript. In this section the performance results of a sophisticated PacketScript application are shown. The application is about caching network packets for TFTP and HTTP data flows. The principle is the same for both protocols. First, a request must be extracted. Second, if a cached response exists for the given request, the response is created and delivered. Third, if there is no cached response for a given request, the request must be passed to the userspace application and its response must be cached. Figure 4.3 depicts such a cache flow. The goal of such an application was to use the advanced features of PacketScript, in order to test and improve them. Furthermore, all protocol logic was developed using PacketScript. The results of the test cases were achieved by measuring the download time using TFTP or HTTP. For both scenarios, 26 different files with exponentially increasing filesizes (starting at 1 byte to 32 megabytes) were downloaded. We repeated each experiment 10 times and calculated the average value. Such caches may also be used to reduce the delay of a network by placing the cache closer to the client. This is a typical scenario targeted by the content distribution network providers. In order to observe such a scenario we used PacketScript to generate an artificial delay while serving non-cached files.

4.2.1 TFTP Cache

TFTP (Trivial File Transfer Protocol) is a simple UDP-based protocol used for the transfer of files. The first packet contains the request. If the TFTP server can serve the request it sends the file in chunks of 512 Bytes, whereas each chunk gets acknowledged by the client. For the experiment we used the Xinetd (version 2.3.14) based TFTP daemon (version 0.17) on the server side. On the client side the Python (version 2.6.4) based tftpy (version 0.5.0) was used for downloading the files. Figure 4.4a depicts the comparison of the average download time for files bigger than two kilobytes, whereas the measurements for files smaller than two kilobytes are shown in Figure 4.4b. The maximum variance is *3%* of the calculated averages when the files are served from the TFTP server, and *2.55%* when delivered directly by the cache. In order to simulate the scenario of a content distribution network, where the cache is normally much closer to the client than the original server, we artificially introduce a delay

between the client and the server. Figure 4.4c compares the cache performance with the delay affected TFTP server performance.

4.2.2 HTTP Cache

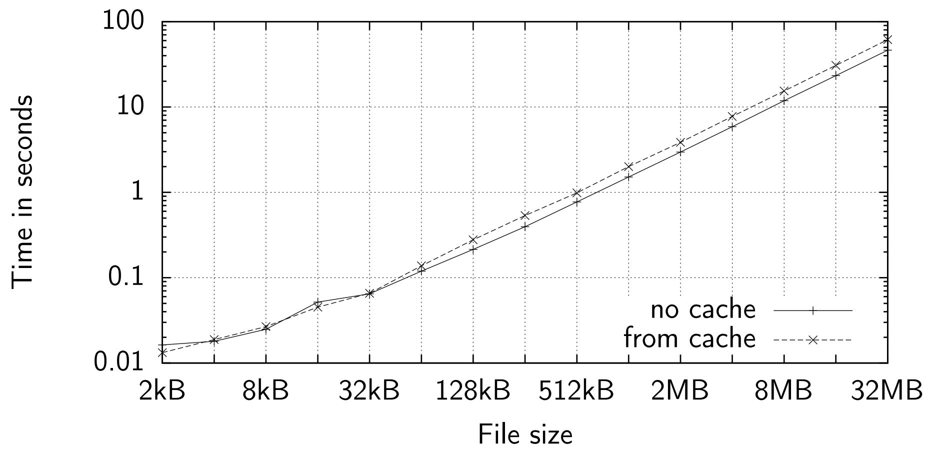
Because HTTP is based on TCP it was necessary to either access a TCP kernel socket or to implement our own TCP implementation. Since this experiment should use as many features of PacketScript as possible, we decided to use the latter. As a result we ported the Python based TCP stack, emerged from the Virtual Network System Project [28] to PacketScript. This network stack was developed with the event-driven network programming framework called Twisted [29]. Some ideas of its reactor module directly influenced the functionality of PacketScript for deferring work. In order to properly dissect the HTTP header, a HTTP protocol buffer was developed. For parsing the HTTP header, we ported parts of the HTTP parsing engine used within the high-performance web server Nginx to the kernel. The parser was then integrated within our HTTP protocol buffer.

For the experiment we used the Nginx web server (version 0.7.62). On the client side the wget tool (version 1.11.4) was used to download the files as well as for measuring the download time. The Figure 4.5a shows two graphs, which compare the average download time for files bigger than 512 bytes, while Figure 4.5b concentrates on smaller files. The maximum variance is *1.38%* of the calculated averages when the files are delivered by the HTTP server, and *0.11%* when served directly from the cache. For simulating the scenario of a content distribution network, where the cache is normally much closer to the client than the original server, we artificially introduced a delay. Figure 4.5c depicts the cache performance with the delay affected HTTP server performance.

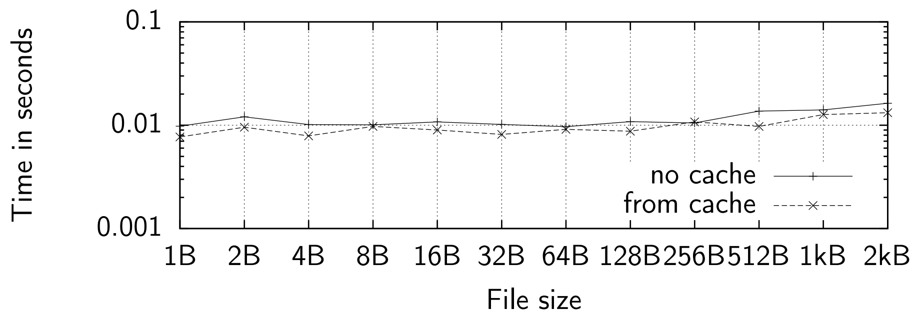
4.3 Discussion / Conclusions

4.3.1 NAT Experiment

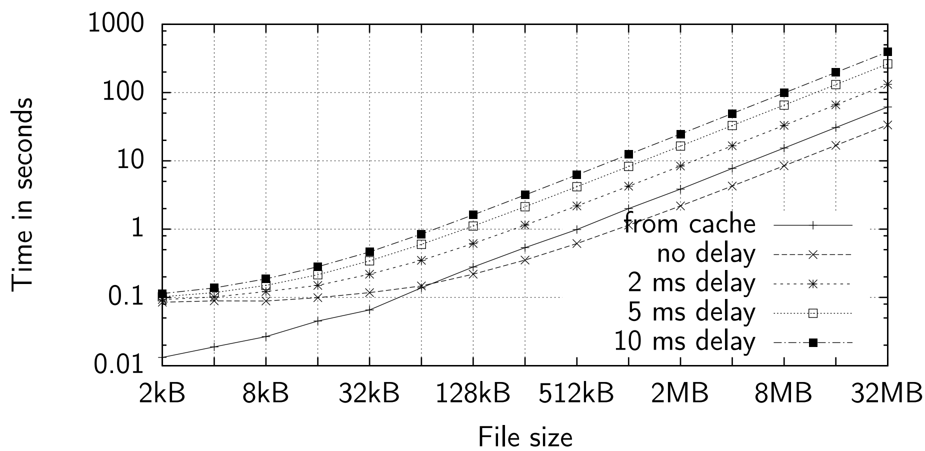
With PacketScript it is easy to create a simple NAT solution, which is able to cope with the high network load produced during the NetPIPE performance benchmark. However, its overall performance is slightly inferior to the NAT configured with the highly optimized Linux SNAT/DNAT targets. Although the differences are very small, they are worth to be discussed.



(a) Comparison of TFTP performances, focussing on file sizes bigger than 2 kB

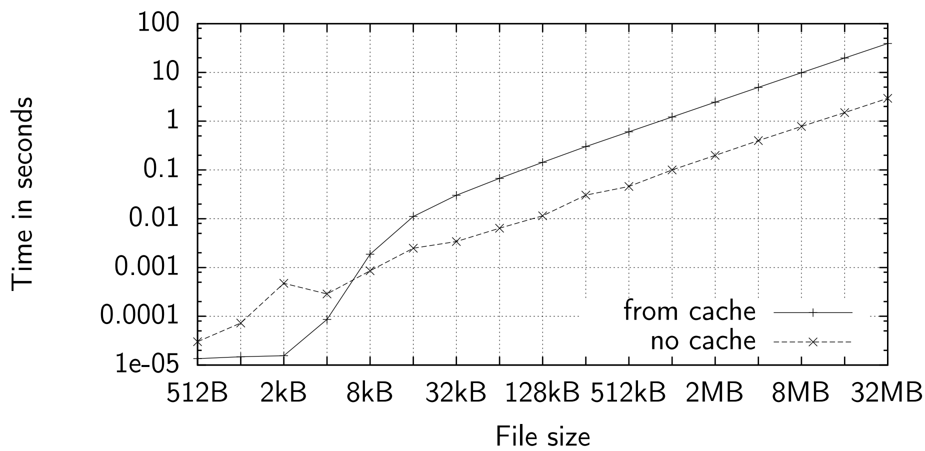


(b) Comparison of TFTP performances, focussing on file sizes smaller than 2 kB

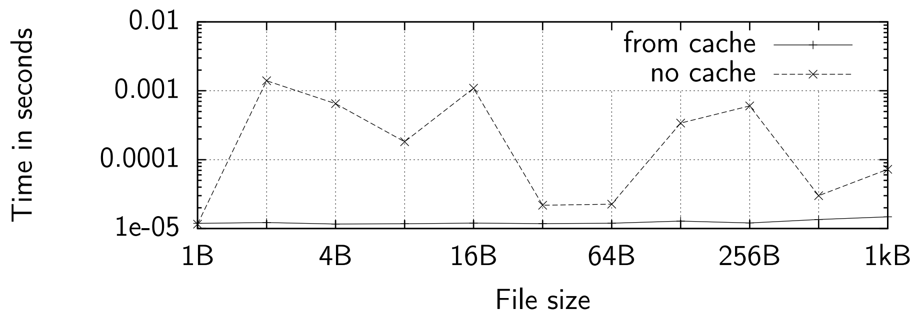


(c) Comparison of TFTP performances for delay affected networks

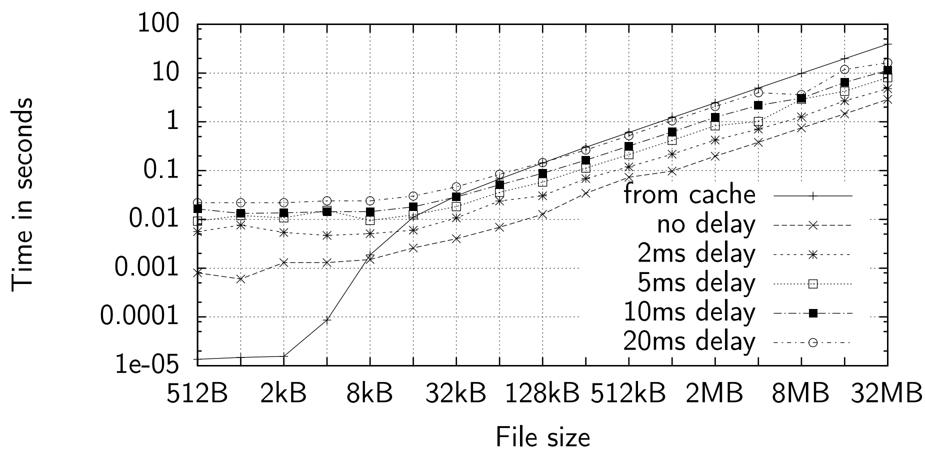
Figure 4.4 TFTP performance test for different file sizes. The files were served from a xinetd based TFTP daemon and from the PacketScript cache. The Figure 4.4a compares both scenarios file sizes bigger than two kilobytes, whereas Figure 4.4b highlights the performance comparison of the smaller file sizes. Figure 4.4c shows how the introduction of a delay may affect the TFTP performance.



(a) Comparison of HTTP performances, focussing on file sizes bigger than 512B



(b) Comparison of HTTP performances, focussing on file sizes smaller than 1 kB



(c) Comparison of HTTP performances for delay affected networks

Figure 4.5 HTTP performance test for different file sizes. The files were served from the Nginx web server and from the PacketScript cache. The Figure 4.5a compares both scenarios for file sizes bigger than 512 bytes, whereas Figure 4.5b highlights the performance comparison of the smaller file sizes. Figure 4.5c shows how the introduction of a delay may affect the HTTP performance.

Both solutions are built upon a complex software stack. On one hand, the Linux SNAT/DNAT uses the connection tracking and NAT subsystem of Linux. On the other hand, the PacketScript NAT relies on the Lua virtual machine. Although the software layers are completely different from each other, both solutions must take care of rewriting the network addresses and recalculating the IP and TCP checksums. For rewriting the network addresses the underlying *sk_buff* structure must be made writable. This process is quite complex, as the whole packet must be copied if the packet is referenced elsewhere. Additionally, fragmented IP packets are reassembled (IP defragmentation) if needed (not the case for this experiment). Defragmentation is expensive for two reasons: First, the fragments must be stored in scarce kernel memory until they are processed by the network subsystem; second, the kernel maintains extra hash tables for their bookkeeping. Modifying the IP and TCP header implies the recalculation of the checksums. As the TCP header checksum also includes the payload data, the calculation time also depends on the block size. Both processes are constrained by the block size, but both NAT solutions use the identical kernel functionality to make a packet writable and for calculating the different checksums. For this reason, these processes will not influence the performance differences between either scenario.

As shown by the results, the network throughputs of both NAT solutions are approximately the same for large block sizes. For large block sizes the overhead generated by each software layer is too small to have a big impact on the throughput. The network saturation graph further confirms this behavior. However, smaller block sizes show an appreciable difference in throughput. This difference is mainly caused by the higher complexity of the Lua VM and the not optimized PacketScript software. Even though the PacketScript NAT solution does not perform as good as the common Linux NAT building blocks, its performance is very close while it offers an entirely programmable and extensible NAT module.

4.3.2 Cache Experiment

Caching packets of the UDP based TFTP was straightforward, since the first arriving packet already contains the request header. Furthermore, a TFTP response can be cached packet-wise, since each TFTP data flow is identical for a given request. In contrast, the TCP based HTTP has to first establish the TCP connection before the HTTP request is sent. Moreover, because of network and TCP dynamics, each data flow of a HTTP response looks different even if the same data gets transfered. This is reflected in the results. As

it can be seen from the results of both scenarios, as soon as PacketScript creates a lot of new packets the performance starts to decrease. In both scenarios the overhead of creating packets with Lua will lead to a performance loss. There are several reasons for such a loss. Typically, the packet creation is done by concatenating several byte arrays holding the different protocol headers. The current version of PacketScript is not very efficient when it comes to concatenating byte arrays. Concatenation is done by dynamically allocating a memory area that is big enough. The byte arrays are then copied to the new memory location. As soon as more than two headers need to be concatenated the performance further decreases. This is the case in the HTTP cache where HTTP, TCP and IP headers must be joined. Therefore, for avoiding unnecessary allocations and copies the *many_to_one* function is applied, which copies the data to a new memory area allocated with *vmalloc* instead of *kmalloc*. *vmalloc* allocates a contiguous memory area in a virtual address space instead of the physical address space. Such an allocator is usually slower than *kmalloc* but it can allocate much larger chunks of memory.

As mentioned before, the HTTP cache utilizes its own TCP implementation, neither optimized nor feature complete compared to the Linux TCP stack. The overhead of creating TCP segments with Lua is even more noticeable since our TCP implementation does not support the TCP window scale option as defined in RFC 1323 [30]. This option is used to increase the maximum receive window size from 65535 bytes to 1 gigabyte. As a result, sending large files is much faster since the sender can send up to the window size unless it has to wait for an acknowledgement and a window update from the receiver.

We used PacketScript to introduce different delays in order to see how our cache solutions perform in a content distribution network scenario. As described in the TFTP results, a network delay of only a few milliseconds will already result in a poorly performing TFTP service. As a result, our TFTP cache performs better than the delay affected userspace TFTP service. However, for beating the performance of a today's web server a few milliseconds delay are not enough.

In order to improve the performance of such applications, the creation of packets needs to be further optimized. Especially, the concatenation of several headers must be improved for reducing large memory allocations and the copying of data. For the HTTP cache it would be very interesting to compare the performance if a TCP kernel socket is used to manage the connections instead of using our own TCP implementation. Additionally, both cache solutions store the packets inside a Lua table, which dynamically grows. The Lua

table implementation is based on hash tables. Depending on the fill level of a table, Lua grows the table and therefore rehashes its entries. This is quite expensive for large tables.

Even though our cache implementations do not perform better than the server software running in user space, they use every feature of PacketScript. The experiments showed that it is feasible to prototype novel network functionality with PacketScript. Additionally, they provide important information about features we have to improve for releasing a next version of PacketScript.

Chapter 5

Conclusion

In this thesis, we presented PacketScript, a framework enabling the rapid development of Linux Netfilter extensions using the Lua scripting language. We developed several plugins for PacketScript in order to provide an object oriented representation of different networking protocols such as Ethernet, IP, ICMP, TCP, UDP, TFTP and HTTP. The development of such plugins is kept easy and can be done using C or even Lua. A Lua script loaded with the *iptables* tool may use such protocol classes for accessing and modifying the fields of the different protocol headers in a simple and object oriented way. Furthermore, PacketScript provides a Lua library for creating and sending network packets. Moreover, it supports the deferring of work using the Linux work queue interface. We showed that a Netfilter NAT extension implemented with PacketScript performs almost as well as using the standard Linux SNAT/DNAT targets. In order to showcase the rapid network prototyping within the Linux kernel, we implemented a caching solution for TFTP and HTTP data streams. For the HTTP cache, we developed a simple TCP stack in Lua, which was used to serve the cache entries. The results of these experiments showed us where our implementation must be further improved.

5.1 Future Work

The way the current version of PacketScript deals with concurrency does not scale very well. Our analysis identified different race conditions as well as how they could be solved: On one hand, we can introduce redundancy and manage several distinct Lua states. This solution implies the development of message passing mechanisms between the Lua states.

On the other hand, we could implement a proper locking solution inside the Lua VM. Further investigations should provide insights what approach to choose but also how it can be realized. Other optimizations should be carried out in how PacketScript creates new network packets. As being discussed, the overhead of creating packets was the main performance bottleneck for the TFTP cache. Especially, the needed concatenation of several network headers must be improved. We think that instead of copying different network headers into one large-enough memory area, the headers can be arranged in a "linked list"-like structure. This will increase the performance, since no data has to be copied anymore. Moreover, there is no need for allocating big memory areas.

Packet sending operations should be further improved as well: The current version of PacketScript keeps a reference to a *sk_buff* structure for providing the necessary link-layer and interface information. This makes the sending of new packets more efficient. However, as long as such a reference is kept, the kernel is not able to free the associated memory resources. Kernel memory is a scarce resource, and thus it should be freed as soon as possible. Another problem of this approach is that we cannot influence the link layer protocol header when sending packets.

The mentioned improvements only focus on the kernel part of PacketScript, but the *iptables* userspace plugin should also be considered for future work. The current version is only able to load one Lua script (one file) per rule, which works great. But, it would be nice to split large scripts into several smaller modules. Lua itself has built-in modularization support, which we cannot use so far. Furthermore, the scripts are loaded using *iptables*, which is supposed to insert rules for analyzing and modifying IP packets. Using PacketScript for mangling packets on lower protocol layers contradicts the intended purpose of *iptables*. For this reason, other tools such as *arptables* for the ARP protocol or *ebtables* for Ethernet frames emerged from the *iptables* project. Therefore, it would be better to constrain the usage of PacketScript according to the different tools. This would simplify the usage for the user, since he would always get an object of the proper protocol class instead of the generic *raw* object. However, such constraints will not satisfy all users. Hence, we should further investigate how PacketScript can be configured to fulfill the needs of the different use cases.

5.2 Implications of Research

Intercepting and manipulating network packets using PacketScript is a revolutionary idea. This is a valuable additional benefit for computer scientists, network engineers, and network administrators. The following paragraphs shortly describe some of the possible use cases:

Content Centric Networking. Content Centric Networking (CCN) is an alternative architecture of computer networks. The main principle of CCN is that members of the network can focus on the data instead of the physical location where that data is to be retrieved from. With the routing of data objects, their caching becomes important. For this purpose a modified, more intelligent data forwarding engine is used. CCN can profit by our approach, as parts of the forwarding machinery can be expressed in Lua using PacketScript. This enables a rapid prototyping development process on an integral part of CCN.

Active Networking. The main goal of Active Networking is to design computer networks, without relying on pre-defined network protocols. The protocols are provided by the members of the network and are dynamically deployed over the network. One big research topic is security inside active networks, as malicious code can significantly damage the network operation. Active networking can profit by our approach, as embedding Lua inside Netfilter brings an additional layer of virtualization that could be used to sandbox delicate operations. Despite the benefits of virtualization, network packets inside active networks could carry Lua scripts that would bring the full power of Lua to active networking.

Protocol Development. Depending on the protocol under development, our approach could be of significant value. On one hand, debugging the protocol becomes easier, as it is possible to access every protocol field on every networking layer. On the other hand, the protocol development could be shifted much earlier from a simulation environment to a productive environment, thus being able to have a working prototype available as early as possible. Furthermore, our approach enables to shape the network traffic in any way we want. Hence, for testing purposes we can introduce packet loss, simulate jitter etc., and still operating in a real environment.

Security Engineering. PacketScript can be used to develop more sophisticated prevention mechanisms for several networking attacks. For the case of a denial-of-service attack,

our approach could easily inspect the application specific payload on a very low networking layer and drop it if needed. Since attack scenarios are changing, the matching rules have to be adapted frequently; in our approach this is as easy as adapting a Lua script.

Data Mining. With PacketScript, a deep-level packet inspection is possible. This can be exploited to extract information for data mining purposes on-the-fly. Data mining becomes a very important tool to transfer "low-level" data into valuable information, which is used in many different areas such as marketing, surveillance, fraud detection and scientific discovery.

Load Balancing. Load balancing in computer networking is a technique to distribute a workload evenly across several computers. With PacketScript it is possible to dispatch the work as early as possible, thus being able to reduce the load from the load balancer itself. Using Lua we are able to develop more sophisticated balancing policies, and still remaining in the kernel space.

Legacy Networking Protocols. In computer networks we have to deal with legacy networking protocols. With PacketScript it is possible to transform a packet, generated by a legacy protocol, to the modern protocol replacement. Of course the other way round is just as much considerable, if connecting legacy protocols with modern solutions.

Bibliography

[1] (2010, Jul.) Netfilter, firewalling, NAT, and packet mangling for Linux. [Online]. Available: http://www.netfilter.org/

[2] R. Ierusalimschy, *Programming in Lua, Second Edition*. Lua.Org, 2006.

[3] D. Brodie. (2010, Jul.) Lua in Kernel. [Online]. Available: http://luak.sourceforge.net/

[4] L. Vieira Neto. (2010, Jul.) Lunatik - Lua (Tied?) In kernel. [Online]. Available: http://lunatik.sourceforge.net

[5] L. Vieira Neto, R. Ierusalimschy, and A. L. de Moura, "Lunatik: a framework for dynamically extending operating system kernels with Lua," Jul. 2010, to be published.

[6] (2010, Jul.) Provide support for dynamic NetBSD kernel extensions using the Lua language - Lunatik/NetBSD. [Online]. Available: http://netbsd-soc.sourceforge.net/

[7] X. Huang, R. Sharma, and S. Keshav, "The ENTRAPID Protocol Development Environment."

[8] S. Wang and H. Kung, "A new methodology for easily constructing extensible and high-fidelity TCP/IP network simulators," *Computer Networks*, vol. 40, no. 2, pp. 257–278, 2002.

[9] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[10] M. Neufeld, A. Jain, and D. Grunwald, "Network protocol development with nsclick," *Wireless Networks*, vol. 10, no. 5, pp. 569–581, 2004.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[12] K. Fall and K. Varadhan. (2010, Jul.) The network simulator (ns-2). The VINT project. UC Berkeley, LBL, USC/ISI, and Xerox PARC. [Online]. Available: http://www.isi.edu/nsnam/ns

[13] D. Ely, S. Savage, and D. Wetherall, "Alpine: A user-level infrastructure for network protocol development," in *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems-Volume 3*. USENIX Association, 2001, p. 15.

[14] M. Jung, E. Biersack, and A. Pilger, "Implementing network protocols in java-a framework for rapid prototyping," in *International Conference on Enterprise Information Systems*. Citeseer, 1999, pp. 649–656.

[15] H. Nguyen and A. Duda, "GateScript: a scripting language for generic active gateways," *Active Networks*, pp. 1–20, 2009.

[16] J. Engelhardt. (2010, Jul.) Xtables-addons. [Online]. Available: http://xtables-addons.sourceforge.net/

[17] R. Russell and H. Welte. (2010, Jul.) Linux netfilter Hacking HOWTO. [Online]. Available: http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.txt

[18] J. Engelhardt. (2010, Jul.) Writing Netfilter modules. [Online]. Available: http://jengelh.medozas.de/documents/Netfilter_Modules.pdf

[19] R. Ierusalimschy, L. De Figueiredo, and W. Filho, "Lua-an extensible extension language," *Software Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[20] R. Ierusalimschy, "Programming with Multiple Paradigms in Lua," *Functional and (Constraint) Logic Programming*, p. 5.

[21] S. Bradner, "RFC 2119: Key words for use in RFCs to indicate requirement levels," Mar. 1997, status: BEST CURRENT PRACTICE. [Online]. Available: ftp://ftp.math.utah.edu/pub/rfc/rfc2119.txt

[22] (2010, Jul.) Threads Tutorial, Locking by Lua. [Online]. Available: http://lua-users.org/wiki/ThreadsTutorial

[23] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers.* O'Reilly Media, Inc., 2005.

[24] A. Eden, E. Gasparis, and J. Nicholson, "LePUS3 and Class-Z reference manual," *Dept. of Computer Science, University of Essex, Tech. Rep. CSM-474, ISSN*, pp. 1744–8050, 2007.

[25] (2010, Jul.) Google Developer Guide - Protocol Buffers. [Online]. Available: http://code.google.com/intl/de-DE/apis/protocolbuffers/docs/overview.html

[26] Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *IASTED International Conference on Intelligent Information Management and Systems*, vol. 6, 1996.

[27] J. Gustafson and Q. Snell, "HINT: A new way to measure computer performance," in *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 2, 1995.

[28] M. Casado and N. McKeown, "The virtual network system," *ACM SIGCSE Bulletin*, vol. 37, no. 1, pp. 76–80, 2005.

[29] A. Fettig, *Twisted network programming essentials.* O'Reilly Media, Inc., 2005.

[30] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP extensions for high performance," May 1992, status: PROPOSED STANDARD. [Online]. Available: ftp://ftp.math.utah.edu/pub/rfc/rfc1323.txt