

A User-Level Development Environment for In-Kernel Network Protocol/Extension Implementations

Alexey Smirnov and Tzi-cker Chiueh
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400
Email: {alexey, chiueh}@cs.sunysb.edu

Abstract—As the Internet architecture evolves towards pushing intelligence to the network edges, new network protocols and functions are routinely added to network edge devices, many of them directly implemented inside their operating system. Examples of such in-kernel additions include firewalling, network address translation, traffic shaping, multi-homing load-balancing, support for virtual private networking, etc. To facilitate the development of these network protocols and extensions, general implementation frameworks, such as Linux’s Netfilter, are created to simplify the process of grafting new functionalities into Linux’s network stack, and to produce logically separate modules that are dynamically loadable. However, Netfilter as an implementation framework does not address the inherent difficulty of programming these protocols and extensions at the kernel level. In this paper, we describe the design, implementation and evaluation of a novel programming environment called *Dusk*, that allows programmers to develop kernel-level network protocols and extensions completely at the user level, and automatically compiles the resulting code into the operating system without any manual modification once the development is finished. We demonstrate the usefulness of this network programming environment by showing how it effectively supports user-level development, execution and debugging of in-kernel Netfilter modules. Finally, the proposed network programming environment is extended to support remote development of new network protocols and extensions, where the programmer’s development environment and the code being developed run on separate machines.

I. INTRODUCTION

Extending the network protocol stack in an operating system with new protocols or functions is a growingly popular exercise as more and more network intelligence is pushed to the edges. Examples of such new network extensions include firewalling, network address translation, traffic shaping, multi-homing load-balancing, support for virtual private networking, etc. Netfilter [1] is an implementation framework for developing network extensions under Linux kernels 2.4 and 2.6. Under Netfilter, one can write a kernel module that can intercept a packet at certain points of the packet processing pipeline, examine and modify it as it sees fit. Through a well-defined programming interface, Netfilter greatly simplifies the process of packet interception and modification.

Traditionally, programmers develop Netfilter modules at the kernel level, and eventually deploy them inside the kernel.

Due to availability of richer debugging facilities and cleaner isolation between the code being developed and the underlying operating environment, user-level code development and debugging is typically much easier than that at the kernel level. As an alternative, programmers can develop Netfilter modules at the user level, and eventually deploy them as user-level modules. This approach shortens the time-to-market value by offering a simpler path of development, but potentially incurs higher run-time overhead due to additional context switching. Ideally, programmers should be able to develop Netfilter modules at the user level, and eventually run them directly inside the kernel without modification. This approach achieves the best of both worlds: the simplicity of user-level development and the efficiency of kernel-level modules. This paper describes the design and implementation of a Netfilter programming environment called *Dusk*¹, that supports exactly this paradigm.

One possible approach to support user-level development of kernel-level Netfilter modules is to replicate the kernel-level API available to Netfilter module developers in the userland. However, because there are hundreds of internal functions inside a typical Linux kernel and they are constantly changing from version to version, this approach is too complex to be practical. Another possibility is to limit user-level programming to only a restricted subset of the kernel-level API. While more feasible, this approach may not be able to support arbitrarily complicated Netfilter modules.

Dusk is a network extension programming environment in which developers program kernel-level Netfilter modules in a way that is identical to user-level code development. However, when a module is completed, *Dusk* automatically converts it into a loadable kernel module that can run directly inside the hosting operating system. As a *Dusk* programmer, she can freely use any kernel-level functions available in the operating system that eventually will host the Netfilter module being developed, and at the same time enjoys all the testing and debugging benefits associated with user-level programming. For example, if a protection fault occurs in the module being

¹Develop at User level and inStall at Kernel level

developed, only the user-level process running the module crashes, but not the kernel.

Dusk is particularly powerful in facilitating the development of programmable network devices (PNEs), which are edge network appliances with limited computational power, for example, firewalls, traffic shapers, VPN gateways, etc. As mentioned in the UC Berkeley OASIS project status report [2], programming and debugging environments for the PNEs are critical to their future success. At the present time, the software for such devices is developed on a PC and then cross-compiled to the target platform. To debug PNE software, the programmer often needs a high-fidelity emulator of the target PNE because the device does not have enough resource to run a debugger on it. With *Dusk*, a programmer can debug a system software for a PNE directly on the target PNE itself using a PC with a standard debugger (such as GDB).

Currently, *Dusk* is based on the GNU C compiler, Linux, and the Netfilter network extension implementation framework. However, we believe the underlying approach is applicable to other combinations of platforms and extension implementation frameworks, such as Linux and Linux Security Module (LSM) interface, FreeBSD and Vnode/VFS interface, Windows NT and NDIS, etc.

The rest of this paper is organized as follows. In Section 2 we review the related work in the areas of operating system extensibility with a focus on the user-level development frameworks for the networking subsystem. We describe the architecture of our system in Section 3. Implementation details are discussed in Section 4. We present the evaluation of our system in Section 5. Finally, Section 6 concludes the paper with a summary of the research contributions and directions for future work.

II. RELATED WORK

A new functionality is added to the operating system by either changing its kernel or by writing modules that are loaded into the kernel dynamically. Virtual file system layer [3], [4], [5], the network stack [6], [1], [7], [8], [9], [10], [11], [12], [13], device drivers [14], [15], the security subsystem [16], [17], and some others are examples of the kernel components that often need to be extended. In a microkernel operating system these components are represented by separate replaceable modules coordinated by a microkernel. Exokernel [18], Chorus [19], Amoeba [20], and V [21] are representatives of the microkernel operating systems approach. Despite the fact that it is always beneficial to run the new components in the kernel mode because of performance reasons, many researchers have developed programming environments that allow user-level extension development.

UFS [3] is a project whose goal is to provide user-level file system extensibility. The interaction between the kernel and the user-level extension is performed through an NFS loopback interface. When using this approach, the kernel assumes that the files being accessed are located on a remote NFS server and uses networking functions to exchange data with what it thinks is a remote machine. These packets, however, are

redirected back to the original machine through the NFS loopback interface to a user-level application that implements the desired functionality. UpcallFS [5] is a stackable file system that can redirect user-specified VFS operations to the user-level using a netlink socket. It can also control which file attributes (such as a file name, full directory pathname, etc.) are passed to the userland.

Device drivers account for a large part of the operating system crashes. The reliability of a device driver can be improved by either isolating it from the rest of the kernel or by moving it to the user space. Nooks [15] is the system that uses address space isolation, function interposition, and object tracking to make existing drivers more reliable. In particular, every extension in Nooks executes in its own lightweight kernel protection domain which execution context has the same processor privilege level, but a limited write access to the address space. Userdev [14] is a system that provides a user-level library of a number of kernel-level functions. Using this library, a programmer can write a kernel-level version of a device driver which will roughly resemble the original kernel-space version.

Alpine [6] is a framework for user-level network protocol development for FreeBSD. It works by moving the network stack kernel source code to the user level and intercepting all network-related system calls performed by the application. Once the user-level network stack completes packet processing, the packet is sent directly to the network interface through a raw socket. This framework allows programmers to write user-level kernel extensions using a kernel interface. However, it required reimplementing a significant number of kernel functions in the userland. The goal of the icTCP project [12] is to give user-level applications control over certain parameters of the TCP protocol such as the congestion window size, the oldest unacknowledged sequence number, and others by providing a set of new socket options. Using these options one can implement various versions of TCP such as TCP Vegas and TCP Nice in userland. Edwards and Muir [11] and Pradhan et al. [13] developed frameworks for user-level TCP as well. The Netfilter simulator project [22] takes an approach similar to that used in Userdev project. The idea is to run the Netfilter part of the Linux kernel source code in user space by providing the necessary subset of kernel functions as a user-level library.

Click [10], [23] and Scout [9] are projects that facilitate development of arbitrary networking extensions by providing a set of basic blocks and allowing programmers to build extensions using these elements. These systems, however, are too fine-grained which increases the learning time of a novice programmer. In particular, these systems do not provide a built-in functionality of a router which is required by most applications. The router component needs to be built up by the programmer for each new application she develops. The Router Plugin [8] project aims at the development of a modular router software architecture in the NetBSD operating system. It allows to bind different plugins to individual network flows. Two additional kernel modules, the Association Identification Unit and the Plugin Control Unit are responsible for matching

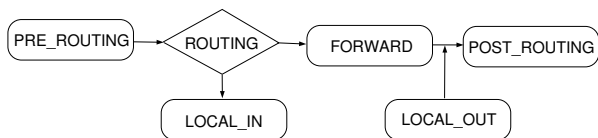


Fig. 1. Netfilter framework defines the following five hooks: `NF_IP_PRE_ROUTING`, `NF_IP_LOCAL_IN`, `NF_IP_LOCAL_OUT`, `NF_IP_FORWARD`, and `NF_IP_POST_ROUTING`.

the incoming packets and forwarding them to the appropriate plugin respectively. FFPF project [24] aims at developing fast, scalable, and flexible packet filters. Extensions written in different languages can be connected to form complex processing graphs.

III. SYSTEM ARCHITECTURE

This section starts with an overview Netfilter architecture. Then we discuss the programming primitives required to support user-level module development. Finally, we describe the components of *Dusk* development framework.

A. Netfilter Architecture

Netfilter defines a number of hooks throughout the Linux network stack. When a module registers itself with Netfilter it provides a callback function. This function is called when a packet passes through a specific hooking point. The callback function receives a pointer to the `sk_buff` structure which corresponds to the packet being processed. This allows the module to examine various fields of the packet and modify them.

The following five hooks are provided by Netfilter: `NF_IP_PRE_ROUTING` — called before a routing decision has been made, `NF_IP_LOCAL_IN` — called for every packet whose destination is the local machine, `NF_IP_FORWARD` — called if packet is to be forwarded to another network interface, `NF_IP_LOCAL_OUT` — called for every packet sent out from the local machine, `NF_IP_POST_ROUTING` — called for every outbound packet before it hits the network interface. Figure 1 illustrates the placement of the Netfilter hooks inside the Linux kernel.

A Netfilter module returns a code to the Linux kernel. This code determines whether the packet will be processed further (`NF_ACCEPT`), dropped (`NF_DROP`), stolen (`NF_STOLEN`), queued (`NF_QUEUE`), or processed from the beginning of the network stack (`NF_REPEAT`). The difference between dropping and stealing is as follows. In the former case, the memory allocated for the packet is freed and the packet is completely wiped out from the system. In the latter case the packet is removed from the processing pipeline, but its content is preserved in memory. Such a packet can be later re-injected into the processing pipeline. Packet queuing is the mechanism in which a packet is placed into a special queue which can be checked by a Netfilter module with a registered queue handler. This mechanism is the basis for writing user-level Netfilter

extensions. When a Netfilter module returns `NF_QUEUE` code, the kernel passes the packet to a registered queue handler.

The Linux kernel provides a simple means of developing user-level Netfilter extensions using queuing mechanism. Module `ip_queue.o` has a queue handler that sends packet's content to userland using a netlink socket. A user process can inspect the packet and return its verdict (`NF_ACCEPT`, `NF_QUEUE`, `NF_STOLEN`, `NF_QUEUE`, `NF_REPEAT`) to the kernel. The packet is then re-injected to the processing pipeline with the returned verdict. The `ip_queue` module provides no means by which a user-level application can modify the content of the packet or call any kernel functions. *Dusk* uses the queuing mechanism to deliver packets to a user-level module as well. To support remote debugging *Dusk* uses a UDP socket instead of a netlink socket to transmit the data.

Packets are not queued by default even if a queue handler is present. Program `iptables` controls packet queuing as well as other parameters of Linux firewall. One can specify the hook type, protocol, source and destination ports, and other parameters which are compared with those of each incoming packet. Only when all parameters match is the packet queued.

B. Programming Primitives

Programs running in the kernel mode are different from programs running in the user land in a number of ways. First, the former can access kernel memory and global kernel variables such as `current` without restrictions. They can invoke other kernel functions directly. If these three components were available to programmers in userland as library functions, they would be able to write user-level kernel modules. Even though, the resulting user-level code would look quite different from the kernel code and manual effort would be required to convert the user-level code into the kernel-level one when the development was finished. The three primitives, however, serve as the basic blocks for the compiler extension that we developed.

The three basic blocks are implemented in the following way. Kernel memory can be read and written using device `/dev/kmem` if the user-level process has root privileges. This file represents the image of the kernel memory. The required address of kernel memory is set by calling `llseek()` with an appropriate offset. A kernel function can be invoked only from the kernel. We solve this problem by providing a new system call `sys_exec()` which takes the address of the kernel function to be executed and its arguments and uses the supplied function address as a function pointer to make a call. The address of an arbitrary kernel function can be obtained from a `System.map` file which is created at the time of Linux kernel compilation. Another way to get the address is to use device `/proc/ksyms`. The value returned by the kernel function is returned by `sys_exec()` to the user process. Global kernel variables are accessed in a similar way. A new system call `sys_kvar()` takes a variable name as a parameter, looks up its address in `System.map` and returns the value read from that memory address to the user-level process. System calls

`sys_exec()` and `sys_kvar()` are implemented in a kernel module which we call the *kernel-level part* of *Dusk*.

The following example illustrates how a programmer can use these building blocks to write userland kernel extensions. Let us assume that a user-level program receives a pointer to `sk_buff` from the kernel-level packet queuing module. In order to access the content of the packet a user-level program will need to (1) create a *mirror user-level variable* of type `*sk_buff`; (2) allocate space for `sk_buff` using `malloc()`; (3) read the data from `/dev/kmem` to the newly allocated buffer using the original kernel address as the file offset. If some changes are performed to the buffer, the data will need to be committed to the kernel memory by writing it to `/dev/kmem` before the function returns or before a kernel function call that uses this buffer is made. In general, the three steps outlined above should be repeated for each kernel-level variable accessed from the userland. Essentially, the *Dusk* compiler performs this task automatically for each memory access when it compiles a kernel-level code into a user-level module.

Dusk was designed with support of remote debugging in mind when the target platform on which the kernel module runs and the developer's machine are connected by a network. In this case file `/dev/kmem` cannot be used to access kernel memory, neither can system calls `sys_exec()` and `sys_kvar()` be used to invoke kernel functions and access kernel memory respectively. *Dusk* solves this problem by connecting the kernel and the user-level components using a UDP socket. Whenever a user-level program needs some kernel services it sends a request to the kernel part including type of the request into the UDP packet. The kernel-level part decodes the request type, performs the necessary actions, and returns the result to the user-level side through the same socket. This solution resembles the remote procedure calls (RPC) mechanism. Figure 2 illustrates the components of *Dusk* framework.

C. Components of Dusk Framework

Dusk framework consists of a kernel-level module and a user-level part which are connected by a UDP socket. We call this socket the *control socket*. The kernel-level part allows the user program to access kernel memory and invoke kernel functions.

Whenever a user-level module registers a new extension with the kernel, an appropriate kernel-level function is called. This function takes an address of a callback function which is invoked by the kernel when the extension is called. The address of the callback function will belong to the user space since this function is a part of the user-level program. However, the kernel cannot invoke a user-level function. Therefore, a proxy kernel-level function should exist for every extension a user-level module can register. This task is also performed by the kernel-level module. The original extension registration function is replaced by *Dusk* compiler with a proxy user-level function. Whenever called this function informs the kernel side about the user program's intent. The kernel part registers a

proxy callback function whose only goal is to send the kernel request over the UDP socket to the user program whenever this function is called. After the user-level program finishes processing the request, it sends the result back to the kernel proxy. The run-time interaction between the user and the kernel components is described in more detail in Section IV. Finally, the user-level side of *Dusk* contains a config file for each target architecture. Using these files, the same user-level program can run on multiple platforms if the appropriately compiled versions of the kernel-level module are loaded on each platform. We describe the components of the platform configuration in more details in Section VI.

IV. RUN-TIME SUPPORT FOR USER-LEVEL MODULE DEVELOPMENT

A kernel module adds functionality to the OS using a set of predefined interfaces. Whenever a module registers a new extension it supplies the address of a callback function to the kernel. We call this process extension registration. Whenever a module is unloaded it unregisters its extensions using extension deregistration functions. A number of different extensions can be registered by a kernel module. In this section we first describe the types of extensions supported by *Dusk*. Extension registration/deregistration process is described in greater detail after that. Finally, user-level handling of kernel requests is discussed.

A. Supported Kernel Extensions

A Netfilter module can register a callback function for any combination of the five available Netfilter hooks. In addition, kernel modules often use kernel threads and kernel timers. We consider timers and threads as special types of kernel extensions as they all use the callback function mechanism. A number of extensions of the same type can be registered simultaneously. For example, the same module can register two Netfilter hooks: `NF_IP_PREROUTING` and `NF_IP_POSTROUTING`. Thus a registered extension is identified by its *type* and its *sequence number*.

Whenever an extensions is called by the kernel it needs to send the kernel request to the user space. Once the request is sent the process running the kernel callback function can either return to the kernel immediately or block in the call back function and wait until a response comes back from the user space. Netfilter extensions are extensions of the former type, whereas threads and timers are extensions of the latter type. Netfilter extensions do not block in the callback function because they are called by software IRQ handler which is used for other purposes as well. Instead, whenever a verdict arrives the packet is reinjected back to the kernel using `nf_reinject()`.

The current *Dusk* prototype does not support kernel threads and timers. However, these extensions as well as any other callback-based kernel extension can be readily implemented in this framework. Table I lists extension registration and deregistration functions that should be proxied to support Netfilter, kernel thread, and timer extensions.

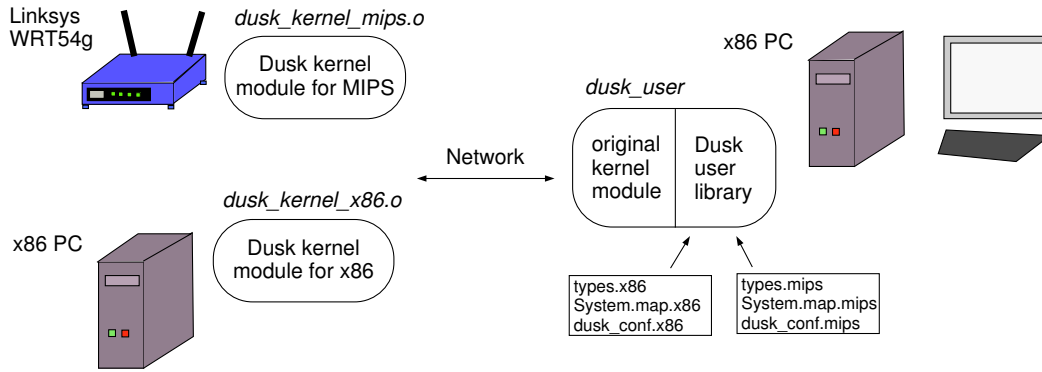


Fig. 2. System architecture of a *Dusk* user-level module.

Extension type	Registration	Deregistration
Netfilter	nf_register_hook	nf_unregister_hook
Kernel threads	add_timer, mod_timer	del_timer
Kernel timers	kernel_thread	—

TABLE I

KERNEL EXTENSION REGISTRATION AND DEREGISTRATION FUNCTIONS.

B. Extension Management

A user-level module registers a new extension by calling the appropriate kernel function such as `nf_register_hook()` for a Netfilter hook. The *Dusk* compiler replaces this function with a registration proxy. The registration proxy performs the following steps. First, it forks a new process that shares its address space with that of the main program and other registered extensions. Then it sends a special registration message to the kernel-side component of *Dusk*. The kernel module registers the kernel extension using one of its proxy callback functions of the appropriate type. Finally it sends an acknowledgment to the user-space module which contains the sequence number of the new extension. Extension type and extension sequence number identify a registered extension uniquely. Figure 3 illustrates the run-time environment created by *Dusk*.

The forked user-level process runs the instrumented version of the callback function from the original kernel module. The extension communicates with the main dispatching process using a message queue. Whenever the kernel calls a callback function, it wraps the request into a UDP packet and sends it to the user-level module. The main process of the module then forwards it to the appropriate extension process using the message queue. Extension type and sequence number are used to find the proper process. When an extension process needs to interact with the kernel it can do so using the second UDP channel between user-space and kernel which we call the *data socket*. All kernel extensions use the same data socket. A semaphore is used to prevent processes from using the socket at the same time. Whenever an extension completes

request processing it sends the result back to the main user-level process first using its message queue. The main process forwards the verdict back to the kernel using the control UDP socket.

One of the main design goals of *Dusk* was to provide a user-level run-time environment that would be close the kernel-level environment. Running different extensions as separate user-level processes resembles the kernel-level execution environment in which multiple kernel extensions can be invoked by different processes at the same time.

When a user-level module is started function `init_module()` is called. When a user terminates the module function `cleanup_module()` is called. In this case each extension sends a de-registration message to the kernel module. The kernel module then calls the appropriate de-registration function.

V. COMPILE-TIME SUPPORT FOR USER-LEVEL MODULE DEVELOPMENT

We have modified the GNU C compiler so that it can instrument the source code of a kernel module in such a way that the resulting program can be run in userland. At the same time, this userland program should be absolutely equivalent to the original kernel module by its side effects such as changes to the kernel data structures and packets being processed. The instrumentation uses the above-mentioned primitives as the basic building blocks.

In particular, the compiler adds the following two functions to each kernel module it compiles: `check()` and `commit()`. The purpose of the former function is to ensure that for every memory buffer that was ever accessed in the program through a pointer there should exist two versions of it: a user-space buffer and a kernel-space buffer. This way we can ensure that both kernel functions and user-level functions have access to the memory buffers used by the program. While the program is running in the user mode, it changes the user-level version of the buffer. When the program is about to switch to the kernel mode, it should synchronize the two versions. Function `commit()` serves this purpose. It is called for each argument

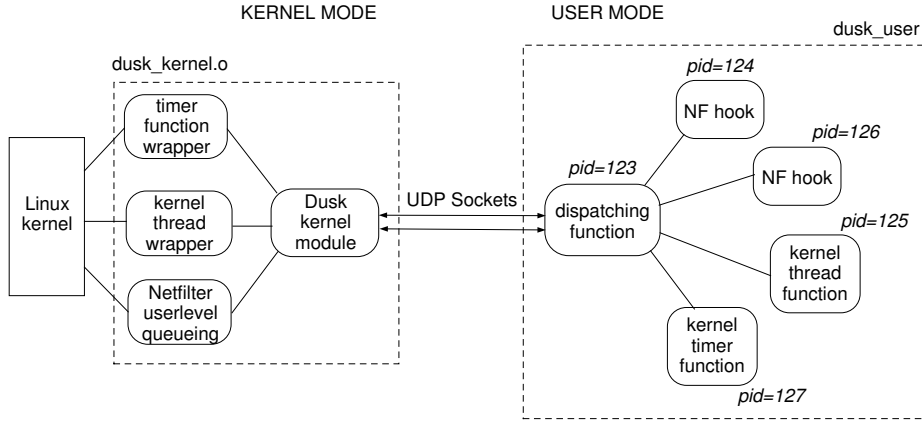


Fig. 3. *Dusk* run-time environment. Module `dusk_kernel.o` is system’s kernel-side component, `dusk_user` is system’s client-side component. Different user-level threads correspond to different registered kernel extensions. The two UDP sockets are the control socket and the data socket.

of a pointer type of a kernel function call. Thus, the kernel will be able to see any changes to its data structures performed at the user level. When the kernel function call returns, the user-level versions of the arguments of this function should be updated by re-reading them from kernel memory. We describe functions `check()` and `commit()` in more detail below.

A. Checking Out Kernel Variables

Function `check()` is applied to every expression that includes a dereferenced pointer, an array element access or an address computation operation (`&`). In all the cases, the `check()` function returns the address of the user-space version of the buffer being accessed. Function `check()` maintains the *address mapping buffer* whose purpose is to store the correspondence between the user-level and the kernel-level addresses. Whenever a pointer is passed to the function, it first searches the address mapping buffer for this address. If it finds a proper record, it returns the user-level address immediately. Otherwise, if the supplied address is a kernel space address (which can be determined by comparing the address with the `PAGE_OFFSET` constant in Linux) then a new user-level buffer is allocated and the data is copied from the kernel space to this new buffer. If the address is a user-level one then a new kernel-space buffer is allocated by calling `kmalloc()` and the data is copied into it. In order to determine the proper length of the allocated buffer, `check()` needs to know the size of the underlying data type. The type information is gathered at compile time and used by `check()` and `commit()` at run time. This information includes the type name, the type length and the number, offsets, and type indices of any pointer references to other types included in this type. For example, type `int*` contains one reference to type `int` stored at offset 0. The two types are considered different if either their length or their list of references are different. Thus, types `int[5]` and `int[10]` are different, but types `int[5]` and `char[20]` are the same type in our system.

For an array element access `a[i]` where `a` is an array of

a known size, `check()` allocates the space for whole array `a` and copies it into this buffer. This is done to ensure that the kernel will have access to the whole array if it is defined in userland. Similarly, whenever an address of a variable is taken (`&a`), `check()` ensures that both the kernel-space and the user-space versions of the variable are created.

If a new buffer was allocated, `check()` stores the new kernel-to-user address correspondence in the address mapping buffer. In addition to the two addresses, the type index and the array index (`i` in case of `a[i]`) are stored in the buffer. For other types of references the array index field is set to zero. These fields are used by `commit()` function when the two versions of the buffer need to be synchronized.

B. Kernel Function Invocation

In addition, the compiler replaces every call to a kernel function (the origin of a function is determined by looking at the header file name where it was defined) with a call to a `void_kernel_func()` or a `int_kernel_func()` which looks up the function’s address in `System.map` and sends an appropriate message to the kernel. Functions `void_kernel_func()` and `int_kernel_func()` are picked up by the *Dusk* compiler automatically based on the return type of the original kernel function. Both functions are linked with the kernel module being compiled. A call made using a function pointer is also replaced by a call to `void_kernel_func()` or `int_kernel_func()` which determine whether the function being called belongs to the user space or kernel space at run-time.

At the time of a kernel function call all its arguments of a pointer type should be the kernel-space pointers because otherwise the kernel will not be able to access them. An exception from this rule are functions that take user-level arguments. The current version of *Dusk* does not support such functions.

The kernel-side buffers should be kept in sync with their user-side counterparts. Function `commit()` serves this pur-

pose. It wraps each argument of a pointer type of a kernel function call. When a data structure image is copied from the user space to the kernel space, it should be taken into account that a kernel function can follow pointers inside this data structure and access other data structures that might have been modified by the user-level program as well. Therefore, the `commit()` process should traverse recursively the references contained in each data structure.

A special care needs to be taken of the case when several array elements are accessed through a pointer. Consider for example a case when a pointer `char *p` is used to access the fifth and seventh elements of a kernel-space array `char *s` and then this pointer is passed as an argument to a kernel-level function. Since the size of the array is not known at compile-time, the `commit()` will only copy the first element of the array omitting others. Therefore, we need to check explicitly whether subsequent elements of the array have been accessed at the user level. Function `get_next_kernel_mapping()` solves this problem by using the array index saved in the address mapping buffer by the `check()` call. It traverses the address mapping buffer and checks whether the difference between the original address and the address of the current record is a multiple of the data structure size. If it is and if the type indices for the two addresses are equal then an array-like reference has been found.

Another relevant issue is committing `union` data types. In this case, several variables of different types can start from the same offset. Checking the type index passed as the function argument and the type index stored in the address mapping buffer can solve this problem. If the two indices are different then a variable of another data type is stored at that location. The corresponding algorithm is presented in Figure 4.

After the kernel function returns, all the user-level images of all data structures that were passed as its argument are updated with the corresponding values from the kernel.

C. Function Prolog and Epilog

Before the user-level Netfilter module finishes processing of a single packet, all the changes that it has done to the kernel data structures need to be committed. *Dusk* counts the dynamic nesting level `depth` of the functions for each extension. Variable `depth` is increased when a new function is called and decreased when it returns. This is done by inserting a call to function `func_start()` to each function’s prolog and a call to function `func_end()` to each function’s epilog. Function `func_start()` increases the value of the `depth` variable, while function `func_end()` decreases it and checks whether it has reached zero. When this condition becomes true, all the user-space memory regions from the address mapping buffer are written back to the kernel.

The `func_end()` function also commits all kernel variables back to the kernel when the `depth` value reaches zero. It also releases kernel memory allocated for local user-space variables while preserving the kernel images of global variables. Therefore, the same kernel image is used whenever a global

Field name	x86 offset	MIPS offset
<code>real_dev</code>	28	—
<code>h</code>	32	28
...
<code>nf_bridge</code>	—	156

TABLE II
DIFFERENCES IN `STRUCT SK_BUFF` LAYOUT ON X86 AND MIPS ARCHITECTURES, LINUX KERNELS 2.4.24 AND 2.4.30 RESPECTIVELY.

variable is accessed by the same or a different extension next time.

Finally, let us consider as an example the instrumentations performed for a single `printk("test")` statement. The internal representation of this expression in the compiler is `printk(&"test")`, that is, the address of "test" is taken. Therefore, the modified compiler will add a `check()` call to the string. Then, the `printk` function will be replaced by `int_kernel_func()`, and finally the function argument will be wrapped into a `commit()` call. The resulting expression is `int_kernel_func(addr_of("printk"), commit(check("test")));`

VI. CROSS-PLATFORM MODULE DEVELOPMENT

Programmable network devices (PNEs) are usually small computers with limited resources used to perform network-specific operations such as network address translation, fire-walling, routing, traffic shaping, and others. For example, Linksys WRT54gs wireless router that we used in our experiments has only 32MB of RAM, 8 MB of flash memory and a 200 MHz MIPS CPU. Because of the low speed of the CPUs and the limited amount of RAM and flash memory available on a PNE, the applications for them are typically developed on a full-fledged PC. Once the development is finished, the applications are cross-compiled and uploaded to the destination PNE.

The WRT54gs device with an OpenWrt firmware [25] runs a Linux 2.4.20 kernel and supports loadable kernel modules and Netfilter. However, the operating system does not contain an application development framework such as a compiler and a debugger. In order to create Netfilter extensions for the WRT54gs device, one has to download the programming environment from the Linksys website which includes the Linux kernel and the GCC cross-compiler. Because of the physical separation of the application development platform and the platform where the applications are running, their debugging becomes even more difficult compared to the debugging of the traditional Netfilter modules. Indeed, the only reasonable way to debug a kernel module on a WRT54gs is to use a lot of `printk()`. If, however, the output is written to the log file such as `/proc/kmsg`, it cannot be used in case of the device crash because the log file is obviously stored in the RAM and not in the flash memory. Therefore, even simplest methods of software debugging such as using `printk()` face many difficulties when applied to PNEs.

```

void *commit(void *addr, int type_ind, int start_ind) {
    int type_size = types[type_ind].type_size;
    void *orig_addr = addr;
    if (is_kernel(addr)) {
        addr = kernel_to_user(addr);
        if (!addr) return orig_addr; /* unmapped kernel address */
    }
    /* traverse this type recursively */
    for (i=0; i<types[type_ind].n_refs; i++) {
        field_addr = (void*)(addr+types[type_ind].offset[i]);
        field_addr = *(void**)field_addr;
        if (!field_addr) continue;
        *(void*)(addr+types[type_ind].offset[i]) =
            commit(field_addr, types[type_ind].ref_type[i], 0);
    }
    /* write original data structure to the kernel space */
    addr = user_to_kernel(addr);
    write_kernel_mem(addr, kernel_to_user(addr), type_size);
    /* check array-like references */
    next_addr = get_next_kernel_mapping(addr, type_ind, start_ind);
    if (next_addr) {
        commit(next_addr, type_ind, (next_addr-addr)/type_size + start_ind);
    }
    return addr;
}

```

Fig. 4. commit() function traverses the data structure recursively.

Because of the differences between the PNE architecture and the PC architecture, several new issues come up when *Dusk* is used to debug Netfilter modules on a different platform. First, although both devices run Linux as their operating system, the kernel version and the configuration of the systems can be quite different. Moreover, replacing a Linux kernel on a WRT54gs device is not as easy as on a PC since doing this requires uploading a new firmware. The differences between the operating systems result in differences in representations of the core kernel data structures such as `sk_buff`. Even though the PNE version of a data structure and the PC version of it have mostly the same set of fields, the offsets of these fields can be different. Table II illustrates the differences between the two structures for Linux 2.4.24 kernel for the PC and Linux 2.4.30 kernel for MIPS. Therefore, the PC-side part of *Dusk* has to be aware of the exact data structure layout in order to generate correct kernel memory read and write requests. The information about the type layout can be obtained by extending the GNU C cross-compiler in the same way as we did with a standard GNU C compiler so that the modified version of the compiler will gather the necessary type information. This information is generated by compiling the Netfilter module by the modified cross-compiler before the debugging is started. The name of the type file is specified in the platform configuration time.

The *Dusk* compiler replaces each structure component reference such as `skb->len` with a call to a special

`get_offset()` function. The function takes the type name and the field name as its arguments and search the target platform types file for the specified type. Then it returns the required field offset using this type information.

The second issue is related to the fact that a PNE has much less memory than a typical PC. This will result in a different values of `PAGE_OFFSET` kernel constant which separates the user-level (addresses less than `PAGE_OFFSET`) and kernel-level (addresses greater than `PAGE_OFFSET`) address spaces. As a result, a kernel-level PNE address can look like a user-level PC address which will make it impossible for the `check()` function to determine the exact location of the data. In order to make all kernel-level PNE addresses look like PC kernel-level addresses, the PC-side function that reads in the kernel data from the PNE increments all addresses in the data that it receives by the difference between the two `PAGE_OFFSETS`. In order to distinguish between a numerical data and an address, this function uses the type information generated by the cross-compiler. Similarly, to restore the original kernel addresses when the data is being sent back to the PNE, the kernel memory write function decrements all addresses by the difference between the two `PAGE_OFFSETS`. The value of the target platform `PAGE_OFFSET` is specified in its configuration file.

VII. EVALUATION OF DUSK

We evaluated various aspects of *Dusk* using a set of the following Netfilter modules: `nfsiff` [26] — an FTP password

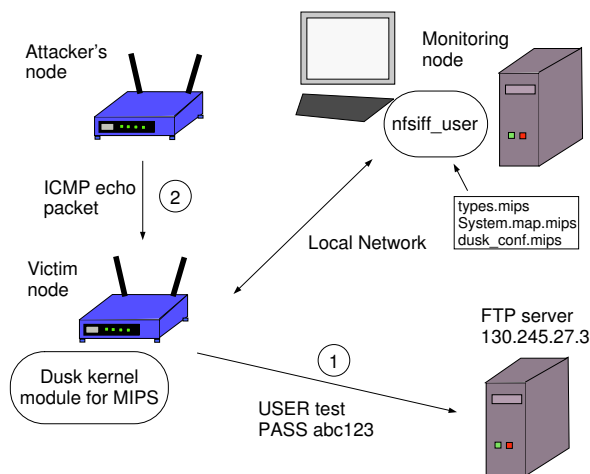


Fig. 5. Hardware setup for the `nfsiff` module. The victim node connects to an FTP server (1). Then attacker’s node sends a special ICMP packet and receives the FTP authentication data from the victim (2). The actual `nfsiff` module runs on the monitoring x86 node which is connected to the victim node through a local network.

sniffer, `lwfsw` [26] — a light-weight firewall, and `ipp2p` [27] — a P2P traffic identifier. We start this section with a description of the test modules. Then we discuss our experiences in user-level module debugging. Finally we present the results of experiments whose goal was to find out the amount of source code instrumentation performed by *Dusk*. These numbers give an estimate of the amount of programmer’s work that has been saved. We used a desktop machine with an AMD Athlon 1.7 GHz processor and 512 MB of RAM working under Linux 2.4.19 kernel as the platform for running the user-level component of *Dusk*. We also had a few Linksys WRT54g wireless routers in our disposal.

nfsiff FTP password sniffer. This module uses two Netfilter hooks: `PREROUTING` and `POSTROUTING`. The goal of the latter hook is to find all `USER` and `PASS` commands in the outgoing traffic and save them in module’s variables. No matter whether the packet contains the desired FTP commands or not, this hook returns `NF_ACCEPT` always. The purpose of the pre-routing hook is to send the grabbed FTP authentication information to the attacker. Whenever the module receives an incoming ICMP echo packet with a special magic code, the packet is transmitted back with the username and password in it. The hook returns `NF_DROP` in this case and `NF_ACCEPT` in all other cases.

We ran this module on both x86 PC and on Linksys WRT54g. Figure 5 illustrates our hardware setup. First, the victim node connects to an FTP server. Then the attacker’s node sends a special ICMP packet that is recognized by the `nfsiff` module. The module saves the grabbed authentication information and sends it back to the attacker. There is also a monitoring x86 PC that is connected to the victim through a local network. In this experiment we run the user-level module on the monitoring node. The *Dusk* kernel-level module

Module name	GCC, s	Dusk, s	Overhead, %
<code>nfsiff</code>	3.0	5.2	73
<code>lwfsw</code>	4.1	4.8	17
<code>ipp2p</code>	4.5	7.4	64

TABLE IV

Dusk COMPILER TIME OVERHEAD. THE TABLE SHOWS STANDARD GCC COMPILER TIME IN SECONDS, THE *Dusk* COMPILER TIME IN SECONDS, AND THE DIFFERENCE IN %.

is installed on the victim node. Its only goal is to receive network packets and send them to the monitoring node. The actual packet processing occurs on the x86 machine. Once the processing is completed, the results are sent back to the victim node. This experiment proved that it is possible to run PNE code on an x86 platform, which is the main goal of *Dusk* project. We ran `nfsiff` module locally on an x86 PC as well.

lwfsw light-weight firewall. This module prevents incoming traffic from reaching a user-level application based on the interface it came through, packet’s port numbers, and its IP addresses. The module uses the `PREROUTING` hook only. The parameters of the firewall are specified through `ioctl` interface in the original module implementation. However, the current version of *Dusk* does not support `ioctl` interface and therefore we set all firewall parameters at the compile-time.

ipp2p P2P traffic classifier. The goal of this module is to recognize the outgoing P2P traffic and to log the statistics about it to the kernel log file. It uses the `POSTROUTING` hook only. The module can identify several UDP and TCP-based P2P protocols.

We ran all three modules successfully on an x86 machine. We also single-stepped through the Netfilter hooks using GDB. These experiments showed that one can debug kernel modules at the user level using *Dusk*. We also introduced a few `NULL` pointer dereference bugs to the test modules. The experiments showed that whenever a user-level module dereferences a `NULL` pointer the corresponding process crashes, but the main process as well as all other processes continue to run. A user can then terminate the module. In all test cases the kernel was not damaged and the system continued to run smoothly. However, whenever a wrong parameter was supplied to a kernel function the whole system crashed. This experiment allowed us to conclude that the current version of *Dusk* protects a developer from `NULL` pointer dereference bugs, but does not protect her from system crashes that happen inside the kernel.

Our final series of experiments aimed at evaluating of how much developer’s work has been saved. The *Dusk* compiler counted the number of instrumented statements of each type, that is, the number of `check()`, `commit()`, and `sys_exec()` calls inserted into the source code. The results are presented in Table III. These results indicate that *Dusk* can indeed save a significant amount of developer’s effort.

We have also measured the module compilation time for the *Dusk* compiler. The results presented in Table IV indicate that

Module name	Lines of code	# check()	# commit()	# sys_exec()
nfsiff	364	71	33	25
lwfsw	222	30	8	8
ipp2p	648	179	68	39

TABLE III

THE AMOUNT OF PROGRAMMER'S WORK SAVED BY THE *Dusk* COMPILER. THE NUMBER OF INSERTED CHECK(), COMMIT(), AND SYS_EXEC() CALLS ARE SHOWN, AS WELL AS THE NUMBER OF LINES OF CODE IN EACH MODULES.

even though the overhead is pretty high, the absolute values of the compilation time are reasonably small because of the small size of the Netfilter modules. Therefore, using *Dusk* should not affect the development process of kernel extension.

VIII. CONCLUSION

In this paper we presented *Dusk*, a framework for developing Netfilter kernel extensions in userland. *Dusk* is designed for extensibility and supports many types of extensible kernel features in userland such as kernel threads, timers, etc. While supporting arbitrary complex kernel modules, *Dusk* does not require programmers to learn any new libraries. It uses the standard kernel API which most kernel programmers are familiar with. The code developed using *Dusk* can be readily compiled into an equivalent kernel module without requiring any changes to it.

We are planning to augment the current prototype with the support of kernel threads, timers, and locks. Another avenue of the future research is adapting *Dusk* to support device drivers. Device drivers account for the biggest part of the Linux source code and are the most frequent reason behind kernel crashes. The previously proposed tools for user-level device driver development either supported only a limited set of kernel functionality or required manual conversion of user-level code into an equivalent kernel-level code. *Dusk* makes it possible to automate this process.

More complex Netfilter modules and device drivers will require better debugger support. For example, one might want to expand a kernel data structure at the debugging time and access the fields that the original module does not access. In this case, the debugger would have to send requests to the kernel module. Also, the programmer might want to access global kernel data structures from the debugger which a standard debugger cannot do either. We plan to add these features to one of the popular debuggers such as GDB or DDD.

We plan to improve the reliability of *Dusk* by extending its protection capabilities. Right now it can only protect a developer from NULL pointer access bugs. We plan to address the problem of deadlocks and invalid function parameters.

REFERENCES

- [1] Netfilter home page, <http://www.netfilter.org>.
- [2] G. Porter, M. Tsai, L. Yin, and R. Katz, "OASIS: Research summary and future directions," 2004.
- [3] D. Mazieres, "A toolkit for user-level file systems," in *Proceedings of the USENIX Annual Technical Conference*, 2001.
- [4] A. D. Alexandrov, M. Ibel, K. Schausser, and C. J. Scheiman, "Extending the operating system at the user level: the Ufo global file system," in *Proceedings of the USENIX Annual Technical Conference*, 1997.
- [5] UcallIFS: Stackable file system development in user land, <http://www.fsl.cs.sunysb.edu/project-upcallfs.html>.
- [6] D. Ely, S. Savage, and D. Wetherall, "Alpine: A user-level infrastructure for network protocol development," in *Proceedings of the third USENIX Symposium on Internet Technologies and Systems*, 2001.
- [7] Y. Gottlieb and L. Peterson, "A comparative study of extensible routers," in *Proceedings of the IEEE Open Architectures and Network Programming conference*, 2002.
- [8] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins — a modular and extensible software framework for modern high performance integrated services routers," in *Proceedings of the ACM SIGCOMM*, 1998.
- [9] D. Mosberger, "Scout: A path-based operating system," Ph.D. dissertation, University of Arizona, July 1997.
- [10] R. Morris, E. Kohler, J. Jannotti, and M. Frans-Kaashoek, "The Click modular router," in *Proceedings of the ACM Symposium on Operating System Principles*, 1999.
- [11] A. Edwards and S. Muir, "Experiences implementing a high-performance TCP in user-space," in *Proceedings of the ACM SIGCOMM*, 1995.
- [12] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Deploying safe user-level network services with icTCP," in *Proceedings of the 6th Symposium on operating systems design and implementation*, 2004.
- [13] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum, "Daytona: A user-level TCP stack," <http://nms.lcs.mit.edu/~kandula/data/daytona.pdf>.
- [14] H. K. Vemuri, "Userdev: a framework for user level device drivers in Linux," Master's thesis, IIT Kanpur, 2002.
- [15] M. Swift, B. Bershad, and H. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.
- [16] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the USENIX Annual Technical Conference*, 2002.
- [17] —, "Linux security module framework," in *Ottawa Linux Symposium*, 2002.
- [18] D. Engler, M. Kaashoek, and J. O. Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the ACM Symposium on Operating System Principles*, 1995.
- [19] M. Rozier, V. Abrossimov, F. Amand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "Chorus distributed operating system," *Computing Systems*, vol. 1, no. 4, 1988.
- [20] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. J. Mullender, A. Jansen, and G. van Rossum, "Experiences with Amoeba distributed operating system," *Communications of the ACM*, vol. 33, no. 12, December 1993.
- [21] D. R. Cheriton, "The V kernel: a software base for distributed systems," *IEEE Software*, vol. 1, no. 2, 1984.
- [22] Netfilter simulation environment, <http://ozlabs.org/~jk/projects/nfsim/>.
- [23] E. Kohler, R. Morris, and M. Poletto, "Modular components for network address translation," in *Proceedings of the IEEE Open Architectures and Network Programming conference*, 2002.
- [24] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly fast packet filters," in *Proceedings of the 6th Symposium on operating systems design and implementation*, 2004.
- [25] OpenWrt firmware, <http://www.openwrt.org>.
- [26] bioforge, "Hacking the linux kernel network stack," *Phrack magazine*, vol. 11, no. 63, 2003.
- [27] IPP2P peer-to-peer traffic identifier, http://rnvs.informatik.uni-leipzig.de/ipp2p/index_en.html.