Bringing Platform Harmony to VMware NSX

Justin Pettit jpettit@vmware.com
Ben Pfaff bpfaff@vmware.com
Joe Stringer stringerjoe@vmware.com
Cheng-Chun Tu tuc@vmware.com
Brenden Blanco bblanco@vmware.com
Alex Tessmer atessmer@vmware.com

Abstract

VMware NSX virtualizes network functionality in a manner analogous to how hypervisors virtualize compute resources. To do this, NSX must faithfully recreate virtual versions of network components, such as switches, routers, and firewalls. As this functionality becomes commoditized, NSX must move "up the stack" to provide more advanced features, such as load-balancers, IDS/IPS (intrusion detection and prevention systems), and DPI (deep packet inspection) for classification.

NSX is designed to work in all types of deployments—even those without any other VMware software. It integrates with ESXi, Linux KVM, and Hyper-V hypervisors; it is even being made to work on systems without a hypervisor, such as containers and third-party clouds. Each of these platforms has its own native forwarding plane. For the best user experience, all of the forwarding planes should provide the same behavior, but the disparate implementations make this difficult in practice. As network functions become more complex and as NSX supports more forwarding planes, both duplication of effort and undesirable diversity of behavior increases.

We propose a new approach to building advanced network functions in NSX. Under this approach, identical code runs on all of NSX's supported platforms. Applications will run at or near native performance, but with better security and identical cross-platform behavior. We demonstrate this by writing a single application to provide DPI functionality that runs in the fast paths of each of NSX's primary platforms: ESXi, Linux, and Edge gateway appliance. We evaluate the performance and correctness of our implementation on the three platforms.

1. Introduction

Compute virtualization simplified the deployment of machines in data centers by disaggregating the operating system from the underlying hardware. Deploying a new system no longer required requisitioning hardware and getting administrators to physically wire it up. The weeks to obtain and deploy a physical machine were reduced to only a few minutes to deploy a virtual machine.

However, virtual machines (VMs) still relied on a physical network. Each virtual machine was given network access via a physical port on its hypervisor, though perhaps limited to a VLAN or restricted in various ways. Thus, a hypervisor's physical location in the network limited the connectivity of all of its VMs. Conversely, any VM's mobility was limited to hypervisors connected to the network segment for its IP address. This also complicated the integration of higher level services, such as load-balancing and security, since traffic needed to be routed through the necessary appliances.

Network virtualization provided by VMware NSX improves this by separating the logical view of the network from the physical. To accomplish this, the network services must be faithfully recreated, usually in the underlying hypervisor. To separate logical and physical addresses, traffic between VMs is encapsulated in tunnels for transmission between *transport nodes*, that is, physical machines that are part of an NSX network.

NSX is designed to be hypervisor-agnostic. Supported platforms includes ESXi, KVM on Linux, Hyper-V on Windows, Linux and Windows VMs in third-party clouds, containers hosted on ESXi, plus a custom-built appliance, called Edge, that allows physical machines to be integrated into an NSX logical network. Development is underway to allow NSX to work with containerized workloads and third-party clouds for which a hypervisor is either not present or inaccessible.

NSX uses the native forwarding plane of each transport node: on ESXi, the NSX Virtual Switch; on Linux and Hyper-V, Open vSwitch; and on the Edge appliance, the Edge virtual switch. These forwarding planes were independently designed and implemented, so each new NSX feature must be implemented at least three times. To provide a seamless user experience, developers attempt to implement identical behavior each time. Unfortunately, issues slip through, because developers on each platform make slightly different assumptions. In addition, behavior naturally differs in cases where NSX relies on platforms' native functionality, e.g. one cannot expect Linux and Windows built-in stateful firewalls to have exactly the same behavior in corner cases.

Sharing code among platforms is a natural way to ensure identical behavior. All of these systems are written in C, so a common C library is a straightforward approach, perhaps using wrappers to expose a common interface to packet buffers and other concepts whose details differ between platforms. Unfortunately, nontechnical issues bar this approach. In particular, the Open vSwitch kernel module on Linux adds special challenges. First, VMware does not control its development—rather, VMware must seek the approval of the Linux networking maintainer to upstream changes, and from distributions such as Red Hat to provide customer support for such changes. Linux maintainers typically reject wrapper-based approaches. Second, the Linux kernel module is licensed under the GNU General Public License (GPL) [6], which adds legal complications to direct code sharing. For these reasons, using a common C library is infeasible in practice.

This paper proposes an alternative way to share code among NSX transport nodes, without abandoning the use of platforms' native forwarding planes. Instead of sharing C code directly, we add support to each platform's native forwarding plane for BPF, a safe, portable bytecode representation. Linux has supported BPF years (see section 2.2); on the other platforms, we port an interpreter. In this model, NSX developers implement a feature once, in C, and compile it to BPF, which runs on all platforms in the same way.

On Linux this approach also sidesteps maintainer and distribution control over development and licensing pitfalls (see section 2.1).

This paper provides the following contributions:

- We show that a single implementation of a high-level network function can run on all of NSX's supported platforms, which reduces duplicated development and ensures identical cross-platform behavior.
- We demonstrate that these new services can run at or near native speeds and with better security.
- We show that these new network services can be implemented without changing the underlying forwarding model of the native networking stack.

The remainder of the paper is structured as follows. The following section provides further background on NSX platforms and on BPF. Section 3 explains the approach we use to provide higher-level services without modifying the underlying forwarding logic. Section 4 evaluates our prototypes running on ESXi and Linux hypervisors, as well as the Edge appliance. Section 5 discusses implementing more complicated services using our proposed model. Section 6 discusses related work, and we conclude in Section 7.

2. Background

This section provides background information to enable the reader to better understand the design.

2.1 Platform Diversity

NSX supports a diverse and increasing array of platforms. Among hypervisors, it already supports ESXi and KVM, Hyper-V is in development, and Xen is often requested. Multiple versions of each of these platforms must be supported. Regarding KVM on Linux, for example, NSX supports Red Hat Enterprise Linux and Ubuntu, each in multiple versions and with different kernels, and new versions and additional distributions are under consideration.

NSX support for container and public clouds is also under development. These environments require a shift from the existing NSX operational model that relies on control over the hypervisor's switching behavior, because container environments do not necessarily have a hypervisor and public cloud operators do not expose the hypervisor switch to their customers. In these systems, NSX integrates with the container or VM operating system, such as Linux or Windows, which can be quite different from running in a hypervisor.

This diversity of platforms is complicated by the fact that each one is supposed to provide identical behavior and functionality under NSX. As the network forwarding plane requires deep integration in the host, it was decided early on that NSX would not require it to be replaced to run. At best, a custom forwarding plane would require additional testing by the various platform maintainers. At worst, the platform maintainer may refuse to allow its use for fear of breaking all network functionality and possibly making the system unstable.

Linux adds special nontechnical difficulty to the mix. VMware directly controls the virtual switch on other platforms, meaning both the source code and the platform itself. On Linux, as a major contributor to Open vSwitch, VMware has substantial influence over its design and implementation, and since Open vSwitch is open source, VMware could "fork" it if it became necessary. The Linux kernel module that Open vSwitch relies on is the real source of issues. VMware is a major contributor to the kernel module, but the upstream Linux kernel networking subsystem maintainer and to some extent the larger community around it must approve of any changes to it. This is often challenging and in several cases these upstreams have refused new features or optimizations [10, 11].

Linux is of course open source itself and therefore VMware could fork the module as well. In fact, the Open vSwitch source already includes source code for the kernel module, which is modified to allow it to compile for multiple versions of Linux, to allow users of older kernels to use Open vSwitch features that normally would require them to upgrade their entire kernel. Thus, in a technical sense, VMware could easily provide new features by adding them to a fork of the kernel module. But Linux vendors do not provide support to their customers who use unapproved kernel modules, and Red Hat in particular has an "upstream first" policy, that is, Red Hat approves only kernel modules whose features have already been integrated into the upstream Linux kernel. In addition, the GNU General Public license required of code in Linux kernel modules would add legal challenges to sharing code between a Linux kernel module and VMware proprietary software.

Platform compatibility issues arise even in platforms for which VMware has control of the entire stack. Before the ESXi switch team introduced the ability to upgrade the forwarding plane, NSX was limited to the capabilities of each particular version of ESXi. There had been great concern about changing the virtual switch, since slight changes in behavior could impact assumptions of other business units. For example, the ESXi virtual switch has particular idiosyncratic teaming behavior that no other virtual switch can exactly reproduce. A change to a different virtual switch would also create additional QE work to retool all virtual switch QE processes to the new switch.

As such, our proposal does not require changing the underlying forwarding plane. Instead, it uses the native forwarding plane for classification, which then directs matching packets to a safe runtime for further high-level processing. While the forwarding plane is different on each system, the use of a runtime can ensure identical behavior regardless of the platform.

2.2 BPF

We approached the idea of a portable packet processing library through BPF, which is a runtime environment based on a virtual machine concept (in the same sense as the Java virtual machine). BPF has been used in networking for decades [9], primarily for filtering network packets; in fact, BPF stands for "Berkeley Packet Filter." The commonly used `tcpdump` program, for example, actually compiles the user's filter, e.g. `host 192.168.1.1` or `port 80`, into a BPF program that selects only the desired packets, and passes it to the kernel. The kernel then executes the program for each packet it receives and copies only the desired packets to userspace. This avoids the cost of copying packets that userspace would discard.

Many operating systems, including BSD, Linux, and ESXi, support BPF for packet filtering. Despite this originally narrow purpose, BPF is actually a general-purpose instruction set that supports arithmetic, logic, memory load and store, branching, function calls, and other common constructs. In recent years, on Linux, this led to adoption of BPF beyond packet filtering, as a general operating system extensibility mechanism. On Linux, BPF programs can now filter system calls, monitor file I/O, analyze live performance, and more. Beyond packet filtering, the most prominent BPF user is probably Chrome, which uses BPF to filter its system calls as part of a sandboxing mechanism [14].

To better support these newer use cases, Linux adds several extensions to classic BPF [5]. These changes allow more complicated programs to be built through the ability to call approved in-kernel helper functions and access persistent data structures called *maps*. Userspace programs can also access maps, which provides userspace a convenient way to monitor and control BPF programs.

Until recently, BPF was usually implemented as a bytecode interpreter. Most BPF programs were short, so the overhead of

interpretation was insignificant. As the use cases for BPF grew and BPF programs grew longer and more complicated, overhead increased. Thus, Linux added the ability to compile, or "JIT," BPF into native machine instructions. JIT-compiled BPF is reputed to run almost as fast as native code, although published numbers are rare; section 4 reports our performance measurements.

BPF programs do not necessarily come from trusted sources. For instance, non-root processes may have permission to capture network packets, and unprivileged processes may filter their own system calls, as Chrome does. Thus, BPF must provide a "safe" runtime environment, that is, it must prevent programs from reading or writing data to which they are not entitled, ensure that programs terminate within a reasonable amount of time, and so on. BPF implementations use a combination of static and dynamic techniques for safety. The dynamic techniques are straightforward: checking, for example, that function arguments are within their supported ranges at runtime.

BPF static checking is more elaborate. It is implemented in Linux by code known as the *verifier*, which evaluates every possible path of the program and tracks its behavior. It ensures that instructions are valid, that jump instructions target valid addresses, and so on. It restricts code to ensure termination, for example by disallowing loops. The verifier also rejects programs that are too large or too complex to verify.

The LLVM toolchain supports BPF as a target architecture, so BPF programs can be written in C and other high-level languages.

These properties of BPF make it a good choice as a way to implement a portable library for packet processing. Implementing efficient L7 features requires running in the packet processing "fast path," which is typically part of the kernel. The performance allowed by JIT compilation and the improved security over native code make BPF an excellent option for extending functionality in the kernel. The next section discusses our larger design for incorporating BPF into cross-platform packet processing engines.

3. Design

For each of the three supported NSX switches, we designed and built a BPF prototype implementation. Service insertion was modeled as a "black box," in which the forwarding plane had no understanding of the functionality provided by the service and only limited information was transferred between them.

This approach simplified integration, since it allowed service insertion to be treated as a simple "action" of the forwarding logic. For packets that needed the service applied, an instance of the appropriate BPF program received the packet and some metadata, such as the logical ingress port. The BPF program returned a status that indicated whether the packet should continue being processed by the forwarding plane or dropped.

All of our designs served the same goals, but because of their diverse platforms they were implemented quite differently. Figure 1 sketches the overall design of each implementation. We discuss the details of each port in the following subsections.

3.1 Open vSwitch on Linux

As shown in Figure 1(a), the Open vSwitch forwarding plane contains userspace and kernel components. The userspace component, *ovs-vswitchd*, contains the forwarding and policy tables. The kernel datapath module is a cache of recently seen flows and their associated actions. Sending packets from kernel to userspace is quite expensive, so ideally traffic hits the datapath cache and stays in the kernel.

This implementation leverages the mature BPF implementation already built into the Linux kernel. The `bpf` system call loads a BPF program into the kernel. The kernel verifies the program, JIT compiles it into native machine instructions, and attaches it to the requested hook point. To prototype BPF for this platform, we added BPF support to Open vSwitch, by adding a new action in the datapath that invokes a BPF program and stores the return value in an Open vSwitch "register," which a later flow matches and actions can use as input.

As discussed in Section 6, work is underway to reimplement the entire datapath in BPF. A goal of that project is to port the BPF runtime to all supported OVS platforms and run the same datapath code. Assuming that project is successful, actions will be implemented as BPF, so no additional work will be required for OVS integration.

3.2 NSX Edge Appliance

Like Open vSwitch, the NSX Edge appliance runs on Linux. However, because Edge uses Intel's DPDK [7] userspace network stack instead of the built-in Linux stack, it cannot use the Linux kernel BPF implementation. Instead, we used the RBPF library, an open source BPF implementation written in the Rust programming language [12]. RBPF includes an interpreter and JIT implementations of BPF, along with a primitive verifier. We used the interpreter in our testing. It allows the client to register helper functions, although it does not yet have built-in helpers for maps.

The LLVM toolchain for compiling C to BPF generates object files in the ELF format commonly used on Linux and other systems. Rust includes an ELF object loader, so our Edge BPF implementation leverages this along with the BCC library for BPF support [4] to dynamically load BPF programs at runtime.

As shown in Figure 1(b), the Edge design consists of DPDK as a bottom layer, with a custom flow cache on top of it, with a port of BSD pf, that is, the BSD kernel firewall, on top of that. BSD pf passes ingress packets through several stages that progress from L2 to L3 to L4 processing and beyond. At each stage the packet may be dropped, or continue, or be held. The Edge BPF implementation inserts a hook after L2–L4 firewall processing.

3.3 NSX Virtual Switch on ESXi

The ESXi in-kernel virtual switch is responsible for forwarding packets. For extensibility, it provides a number of hooks, called *IOChains*, that allow functions to be introduced at runtime that are called at various stages of packet processing. The NSX distributed firewall, for example, uses these hooks to integrate into an ESXi deployment.

The TCP/IP stack in the ESXi kernel contains a BPF interpreter, but it only works with vmkernel network interfaces, that is, it is not on the path of packets to VMs. Also, it is intended only for packet filtering and lacks features required for general-purpose work. For production use it might make sense to use a single engine for both purposes, but for proof-of-concept purposes we introduced a second BPF engine.

For simplicity, our BPF implementation on ESXi does not support loading a BPF object file directly into the kernel. Instead, we use a utility to convert an object file into a C header file and then statically compile that directly into the switch driver.

As shown in Figure 1(c), we hooked the BPF runtime into an IOChain. For traffic coming into and out of a virtual interface, the packets are passed to the attached advanced service.

4. Evaluation

We evaluated the performance and correctness of our prototype using a BPF program that we designed as our test case. This test case was run unmodified on each of the NSX supported platforms. The following sections describe the test case itself, then our performance and correctness tests and the results.

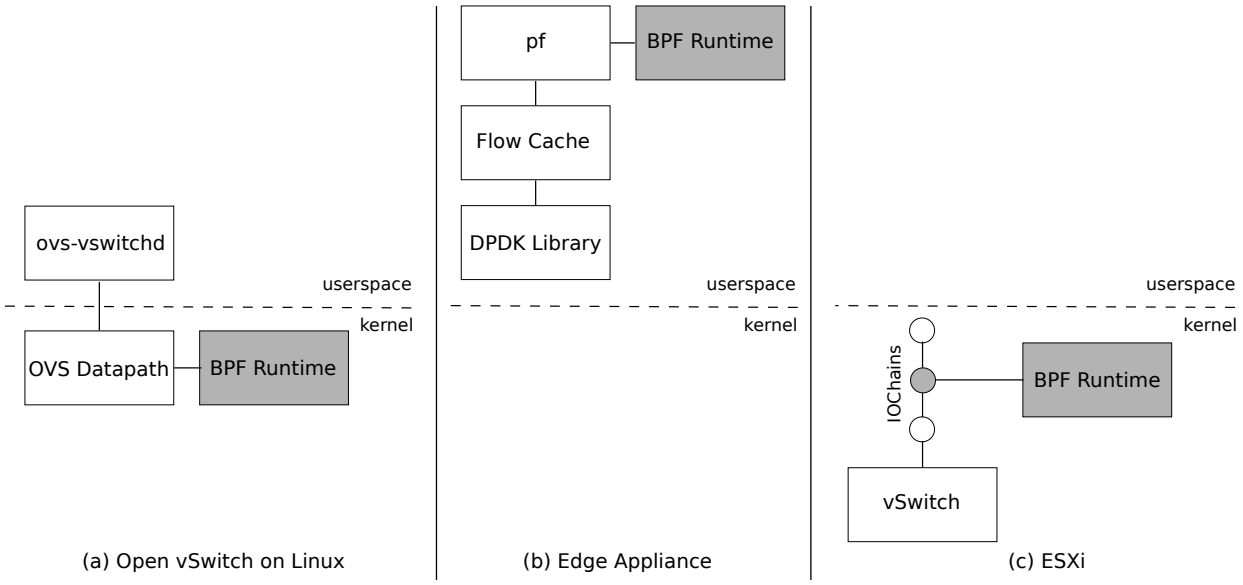

Figure 1: Service insertion was achieved by porting the BPF runtime to each of NSX's supported platforms. (a) On Linux, the Open vSwitch forwarding plane contains userspace and kernel components. We modified the OVS kernel datapath to call into the Linux kernel's existing BPF stack. (b) The Edge appliance is built with DPDK, which means all packet processing occurs in userspace. For the prototype, a custom-ported BPF runtime is called by the pf component just after the L2-L4 firewall hook. (c) The ESXi forwarding plane executes entirely in the kernel. We attached the BPF runtime to an IOChain, which the vSwitch provides as an extension mechanism.

4.1 Test Case

As a test case for for our prototype, we implemented in BPF a *deep packet inspection* network function. Deep packet inspection, called DPI for short, attempts to identify the protocols used in network connections by examining the data transferred inside TCP and UDP payloads. Identifying protocols via DPI is more computationally intensive than by looking at TCP or UDP port numbers, but it is also more accurate, especially as new protocols tend increasingly to be layered on top of HTTP on port 80. DPI is commonly deployed in enterprise networks [1] to allow different types of traffic to be treated differently; for example, voice-over-IP traffic might be routed to minimize latency, or social networking traffic might be de-prioritized or dropped. An upcoming release of NSX will include a DPI feature; our prototype is an independent implementation.

Our particular test case identifies the URL in an HTTP request, drops requests against domains that appear in a blacklist, and passes through other packets. The blacklist consisted of `yahoo.com`, `facebook.com`, `example.com`, and `youtube.com`. The non-blacklisted URLs were the home pages of `vmware.com`, `google.com`, and `slashdot.org`.

Our DPI implementation is a proof of concept, that is considerably simplified from what would be required in production. A production-quality DPI implementation would reassemble TCP packets to view the stream of data; our DPI prototype only examines the payload of individual packets in isolation. A production DPI implementation would allow the protocols understood to be dynamically configured at runtime; our DPI prototype supports only extracting the domain from an HTTP request. We believe that our experiments remain valuable, even with these caveats.

4.2 Correctness

A program that is fast but not correct has little value. Our correctness test seeks to demonstrate that our BPF test case behaves the same way on all of our platforms, by running it on each platform and comparing the results.

Using the `wget` HTTP client, we retrieved a variety of URLs, both on and off the blacklist, on each platform. We successfully verified that fetching URLs on the blacklist timed out and that other URLs could be retrieved retrieved normally.

4.3 Performance

To measure performance, we used a packet generator to send a recorded collection of HTTP packets that contains a mix of packets to be be accepted and to be dropped, at maximum line rate for a 10-Gbps link through each switch under test. We generated load this way because it was easily reproducible and because generating such a high load with regular web clients and servers would require much more hardware. We report the results as the number of packets and bits per second successfully processed, which is the sum of packets sent on an output port and packets dropped due to blacklisting. None of the implementations could keep up with the full 10-Gbps test load, so the reported results include only the packets that were actually processed.

Performance-wise, our main goal is to demonstrate that BPF performance is not significantly worse than conventional alternatives. Thus, in addition to implementing DPI in BPF, we implemented a "native" version of the same functionality in the Open vSwitch kernel module for Linux. We measured both implementations on the same test data, using a single CPU core and a single NIC queue. Table 1 reports these measurements, plus "baseline" numbers without DPI running at all. The measurements show that DPI via BPF, with JIT compilation, exacts a 5% penalty in terms of packets per second and 6% in bits per second for our test set.

We were not able to take performance measurements for the other platforms due to time and resource constraints. In our prototype, these platforms use a BPF interpreter rather than a compiler (RBPF does include a JIT, but we were not able to use it in our

Measurement	Mpps	Gbps
No DPI	1.30	3.57
DPI, native implementation	1.28	3.53
DPI, BPF implementation (JIT)	1.21	3.31
DPI, BPF implementation (interpreted)	1.12	3.04

Table 1: Open vSwitch on Linux throughput without DPI and with two DPI implementations, in millions of packets per second and gigabits per second.

testing). When we test DPI in BPF on Linux with JIT compilation disabled, we see about 15% total penalty, as shown in Table 1. We believe that it is reasonable to assume that these platforms would see about the same relative penalty. Of course, for production use, one would implement a JIT.

5. Discussion

Prototype Simplifications. The Linux prototype benefited from having a full extended BPF implementation at its disposal in the kernel. As noted in Section 3, we ported different BPF runtimes to ESXi and the Edge appliance. These runtimes had various drawbacks, such as license issues, missing features (e.g. lack of maps that limited the complexity of the test case), incomplete or missing verifier, and no JIT compilation to native machine instructions. For production, a single BPF runtime for the non-Linux platforms could be maintained to address these issues.

Depending on the hardware and operating system, identical packets could be presented differently to the BPF program. For example, the platforms differ in their treatment of 802.1Q VLAN headers: ESXi and Edge and old versions of Linux include the VLAN header in packet data, but recent versions of Linux remove it and instead provide it as metadata. We worked around this difference by making the BPF program tolerate both, which allows a single program to work on old and new Linux as well as ESXi and Edge, but it could also be addressed by using slightly different programs on each platform.

BPF Limitations. BPF programs perform well and can be stateful, but restrictions on their complexity, imposed by the verifier on each platform, could arise as an issue at some point. On Linux, for example, the verifier restricts each BPF program to about 4,000 instructions—although multiple programs can be chained—and, to limit the verifier's own runtime, it also restricts program complexity. This means that some sophisticated programs may not be appropriate for BPF implementation; for example, implementing a TCP-terminating protocol stack in BPF may be beyond the sensible limit. In such a case, a native library could be provided that the BPF program accesses through helper functions. This is also a reasonable way to give BPF programs access to functionality that cannot reasonably be implemented without loops, such as regular expression search.

Security Benefits. Datapath functionality implemented in BPF may have a security advantage over functionality implemented directly in C, because the BPF verifier and runtime environment prevents some kinds of security vulnerabilities such as buffer overflows. Exploitable vulnerabilities of the kind that BPF would prevent have been found in virtual switches [13]. Figure 2 provides an example of an unsafe program and the verifier refusing to load it.

Partner Services. VMware partners can extend NSX functionality. For example, the Palo Alto Networks firewall virtual appliance can be used as an alternative to NSX's built-in firewall [2]. Passing all traffic through a virtual appliance introduces unacceptable overhead, so NSX provides a mechanism to limit the packets to divert, currently with limited flexibility. Replacing it by a BPF program could allow third parties to safely, efficiently, and portably enforce policies. For example, for a DPI service, the virtual appliance only needs to look at a particular flow until its protocol is identified.

```
>>> import bcc
>>> b = bcc.BPF(text="""
... BPF_HASH(foo, u64, u64);
... int kprobe__finish_task_switch(void *ctx) {
...     u64 key = 0;
...     u64 *val = foo.lookup(&key);
...     (*val)++;
...     return 0;
... }
... """)
bpf: Permission denied
0: (b7) r1 = 0
1: (7b) *(u64 *)(r10 -8) = r1
2: (18) r1 = 0xfb729d80
4: (bf) r2 = r10
5: (07) r2 += -8
6: (85) call 1
7: (79) r1 = *(u64 *)(r0 +0)
R0 invalid mem access 'map_value_or_null'
```

Figure 2: The BPF verifier blocks loading potentially dangerous code. In this example, the static analyzer in the verifier denies loading the program because `val` may be null. The program would have succeeded in loading had the code checked that `val` was not null before attempting to dereference it.

Other Uses of BPF. This paper concentrates on the application of BPF for networking in NSX. Outside VMware, BPF is already used in areas including tracing, monitoring, security, and hardware offload [3]. As VMware introduces cross-platform products, these use cases may also become relevant to these new products.

6. Related Work

Our proposal adds BPF to OVS as an extension to the existing Linux kernel module. We are also exploring the more radical approach of replacing the entire OVS kernel module by a BPF program (or collection of programs), as a way of solving the nontechnical kernel module maintenance issues described in Section 2.1. Additionally, this could improve portability across OVS supported platforms by using an identical datapath that only requires a ported BPF runtime. Currently, a separate datapath is maintained for each platform, and the features supported by each datapath varies widely. An accompanying OSR article [15] describes this proposal in more detail.

We propose a "black box" approach in which each service is individually responsible for parsing the packet and reassembling the data. For production we may want to consider an approach closer to that of SoftFlow [8], which shares more state between the base forwarding plane and the services. Classification and reassembly are expensive operations, which consume the majority of forwarding plane CPU time in our experience, so reducing redundant computation could yield large benefits.

7. Conclusion

This paper discussed compatibility issues we've seen in maintaining multiple different forwarding planes in NSX and a possible approach to providing higher level services using a portable runtime. To demonstrate this, we ported the runtime to all supported NSX platforms. We then wrote a program to do deep packet inspection and ran it unmodified on each of the platforms. We were able to demonstrate this as a viable approach to building advanced network services with better portability, improved security, and similar performance to native-built functionality.

References

[1] Deep packet inspection. https://en.wikipedia.org/wiki/Deep_packet_inspection.

[2] VMware NSX with next-generation security from Palo Alto Networks. https://www.paloaltonetworks.com/resources/techbriefs/vmware-nsx-solution-brief.

[3] Use cases: IO Visor project. https://www.iovisor.org/technology/use-cases, May 2017.

[4] Brenden Blanco, Brendan Gregg, Sasha Goldshtein, et al. BPF compiler collection (BCC). https://github.com/iovisor/bcc.

[5] Jonathan Corbet. Extending extended BPF. http://lwn.net/Articles/603983/, 2014.

[6] Free Software Foundation. GNU general public license, version 2. https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html, June 1991.

[7] Intel et al. DPDK: Data plane development kit. http://dpdk.org/.

[8] Ethan J Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. SoftFlow: A middlebox architecture for Open vSwitch. In *2016 Usenix Annual Technical Conference (USENIX ATC 16)*, pages 15–28. USENIX Association, 2016.

[9] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, volume 46, 1993.

[10] David Miller. [GIT net-next] Open vSwitch. https://www.spinics.net/lists/netdev/msg291696.html, August 2014.

[11] David Miller. net: Add STT support. https://www.spinics.net/lists/netdev/msg314619.html, January 2015.

[12] Quentin Monnet et al. rbpf: Rust (user-space) virtual machine for eBPF. https://github.com/qmonnet/rbpf.

[13] Ben Pfaff. CVE-2016-2074: MPLS buffer overflow vulnerabilities in Open vSwitch. https://mail.openvswitch.org/pipermail/ovs-announce/2016-March/000222.html, March 2016.

[14] Julien Tinnes. Introducing Chrome's next-generation Linux sandbox. http://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, September 2012.

[15] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. Building an extensible Open vSwitch datapath. In *ACM SIGOPS Operating Systems Review*, 2017.