Porting Kernel Code to Four BSDs and Linux

Craig Metz
ITT Systems and Sciences Corporation

Porting Kernel Code to Four BSDs and Linux

Craig Metz
ITT Systems and Sciences Corporation
`cmetz@inner.net`

Abstract

The U.S. Naval Research Laboratory develops and maintains a freely available IPv6 and IP Security distribution. All of the software builds and runs on BSD/OS, FreeBSD, NetBSD, and OpenBSD, and a growing portion of the software builds and runs on Linux. Each of the four BSDs has evolved significantly from their original 4.4BSD-Lite ancestor, and increasingly more of that evolution is along divergent paths. Linux shares no significant ancestry with the BSDs, but is still a POSIX system, which means that many of the same high-level facilities are available even though their implementation might be completely different.

This paper discusses many of the differences and many of the similarities we encountered in the internals of these systems. It also discusses the techniques and glue software that we developed for isolating and abstracting the differences so that we could build a significant base of system code that is portable between all five systems.

1 Introduction

1.1 History

The U.S. Naval Research Laboratory's IPv6+IPsec distribution [1] began its life in 1994 on BSD/386 1.1, which was a 4.3BSD "Net/2" system. When BSD/386 2.0, which was based on 4.4BSD-Lite, became available, we moved the code to that system. This was a straightforward change, as the differences in the parts of the two systems that our code interfaced with were small. We then added support for "real" 4.4BSD/sparc. At this point, we ran into our first two experiences with maintaining the same code in different kernels. First, we had to separate our code into parts specific to each and parts shared between the two. Second, we found slight differences in the way the systems did things and had to add some **`ifdef`** statements to our "shared" tree to cope with these. Still, these were virtually identical systems. Because a large portion of our code resided in the **`netinet`** directory and the two systems were virtually identical, we decided to put our modifications into the 4.4BSD-Lite2 **`netinet`** (IPv4) code and make the needed changes to port that into BSD/386 and 4.4BSD/sparc.

In 1995, we made the first jump to a radically different system. Although most of the research community used BSD, Linux was also an interesting system, so we decided to attempt to port a component of our software to Linux. One component that was somewhat unique to the NRL software was our PF_KEY[2] interface and implementation, which also had the important feature of being a fairly self-contained module. PF_KEY communicates with user space as a new sockets protocol family and with the rest of the kernel through function calls we defined. We felt that it would be useful to have the PF_KEY implementation running on Linux even if the rest of our code never did, and that it was a reasonable part of the software to attempt to port. We split our PF_KEY implementation and our debugging framework into a OS-specific parts (such as the sockets interface code) and common parts (such as the actual message processing code). With some help from the Linux community and Alan Cox in particular, we ported our 4.4BSD PF_KEY implementation to to Linux in about three months.

In 1996, we added support for NetBSD, another 4.4BSD-Lite derived system. The NetBSD team had made a number of changes versus 4.4BSD-Lite that we had to add **`ifdef`**s to handle. NetBSD had also added a number of new IPv4 features that 4.4BSD-Lite didn't have; because we were using a common 4.4BSD-Lite **`netinet`** implementation, those features were all removed when we replaced that code.

As time went on, we dropped support for the "real" 4.4BSD/sparc because almost nobody actually had a copy of it and because NetBSD/sparc was a freely available system that worked better. Both BSD/386 (renamed to BSD/OS) and NetBSD continued to evolve, and we had to put more and more `ifdefs` into our code to cope with their changes from 4.4BSD. It became clear that we were actually spending almost as much effort shoehorning the original 4.4BSD-Lite `netinet` code into the newer version of these systems as it would take to maintain our changes to the `netinet` code in each system's native implementation, and throwing away the native systems' changes was a problem for some users.

In 1997, we built an implementation of the second version of our PF_KEY interface. We chose to build this implementation as a ground-up rewrite. Instead of building this new implementation on one system and "porting" to the others, we actually developed this code simultaneously on BSD/OS, NetBSD, and Linux. By doing this, we were able to find and resolve OS dependencies quickly, and we were able to take advantage of the debugging strengths of each of these systems. This was also an opportunity to do many things differently, designing for portability based on our previous experiences rather than adding portability as an afterthought.

In late 1997, BSDI chose to integrate our code into their source tree for the upcoming BSD/OS 4.0. BSDI actually did the work of integrating our changes into their `netinet` code, which meant that half of the work of integrating our code into the native `netinet` code of our supported systems was done. We also decided to expend more effort on becoming a good choice for integration into other systems, which meant adding new ports to OpenBSD 2.3 and FreeBSD 3.0 and integrating our changes into the native `netinet` implementations of each system.

The OpenBSD port was surprisingly straightforward because of its shared ancestry with NetBSD; most of the differences between 4.4BSD-Lite and NetBSD could also be found in OpenBSD. The majority of the work we ended up doing for this port was integrating our code into the OpenBSD `netinet` code, which then got us most of the way to also having our code integrated into the NetBSD `netinet` code.

The other port we added was to a snapshot of FreeBSD 3.0. This port was much more difficult, because FreeBSD had made many more substantial changes from 4.4BSD-Lite than the other BSD systems we supported. Working with a snapshot turned out to create problems, too. The system had bugs that we ran into, and many of the system's debugging facilities didn't work at all. We also had more work to do when we moved from the snapshot to the real 3.0 release, as there were significant new changes versus 4.4BSD-Lite that we had to handle.

As both the OpenBSD and FreeBSD ports were starting, we had some level of support for five operating systems and we were maintaining much more OS-specific code than we were before because we were modifying each of the BSD systems' native `netinet` code. We decided to do a major reorganization of our source tree, separating it into seven modules: an OS-specific piece for each of the five systems, a BSD-common piece shared between all of the BSD systems, and a common piece shared between all of the systems. With some added scripts and tools, we were able to create an organization of "overlays" such that the common pieces could be symlinked to appear in the OS-specific directories, and such that the more-specific pieces could override the less-specific pieces. This new organization made it much easier for us to maintain our software on all of these systems while trying to balance duplication of effort with close integration into the supported systems.

While we were doing the ports to OpenBSD and FreeBSD, we were also rewriting large portions of our IPsec implementation. Because we knew that it was a direction we wanted to eventually move to, we built a framework to allow the new code to be ported to Linux. After all of the BSD ports and the rewrite of the IPsec code were finished and released, we began the task of porting our IPsec code to Linux, which is currently a work in progress.

1.2 Observations

In the course of building our software and adding support for all of these systems, we learned many things about how to build and maintain portable kernel code. Many of the lessons that we learned didn't have to be learned the hard way. Large blocks of code are shared between the various BSD systems, especially code for new networking features and device drivers. There are also some cases of software that had been ported from BSD to Linux (for example, the NCR SCSI driver) and from Linux to BSD (for example, the GPL x87 math emulator). Perhaps the best known example of porting kernel code is the BSD network stack, which can be seen running in all sorts of systems that are definitely not BSD UNIX. However, we are not aware of anyone who has ported code to the various BSD and Linux systems and really documented what they discovered while doing it.

The rest of this paper has two major parts. In the first part, we will document our high-level discoveries about building and maintaining portable kernel code. These are general observations about how to structure such code, how to go about building such code, and – possibly more important – what we discovered that you should *not* do. In the second part, we will document many of the specific mechanisms we developed for making code portable. These range from simple wrapper macros to major new compatibility data structures.

2 High-Level Discoveries

2.1 Should You Port It?

Probably the most important thing to consider before getting to far into thinking about porting kernel code is whether something really should or shouldn't be ported.

Given enough time and effort, any piece of system code can be ported to any other system. However, for many pieces of code, that will mean either porting a large chunk of the original OS with it or otherwise dramatically changing the target OS to be more like the original OS. Generally speaking, this is not a good thing to do.

Some code is so dependent on the system that the problem that the code solves might not exist in other systems or the solution approach might not work in other systems. Or maybe someone else is already doing a good enough job of solving the problem on another system that you don't need to port your code there.

This is a lesson we learned, though not the hard way. While we were getting our IPv6 implementation for 4.4BSD and NetBSD more stable, Pedro Marques and a few other developers were working on an IPv6 implementation for Linux. An IPv6 kernel implementation requires extensive changes to the existing system to generalize IPv4-dependent code, and these changes are very system dependent. Since that represents the majority of the effort required to build an IPv6 implementation, it didn't make sense for us to port our kernel implementation to Linux, since we would have to do most of the work from scratch only to be duplicating the effort of another group.

Note also that some systems are architecturally similar and some are very different, and it makes a lot less sense to try to port kernel code between very different systems than very similar systems. For example, both 4.4BSD and Linux are UNIX-like systems, and they both have sockets-based IP network stacks. So, while the implementation of network code might be very different between the two systems, there are still a lot of high-level similarities because they are constrained in their architectures by the standardized interfaces they present (POSIX, sockets, IP). In contrast, porting code from the Linux kernel network stack to the Win95 kernel network stack might be a lost cause because the systems don't share enough architectural similarities.

Another important note is to observe the distinction between kernel space and user space. While it is sometimes possible to take a module written for kernel space and move it to user space or vice versa, most code written for use in kernel space is that way for good reasons. As a general rule, don't try to move code through the kernel/user barrier.

2.2 Portability Techniques

Writing really portable code is not hard, but it is tricky. Almost all of the same approaches, tricks, and traps that you encounter porting user-space code applies to kernel-space code, too. One of the unfortunate problems resulting from increased standardization of systems is that a lot of people have never ported code from one radically different system to another, so many people really don't know how to take code and make it portable.

As a general rule, a good approach to making code portable is to add new abstractions. For substantially similar operations that just work a little differently on different systems, this approach works really well – replace the concrete code with an abstract macro that expands into different code on different systems. You might also consider giving up some of the flexibility that particular systems give you. For example, BSD systems have a function `malloc(size, type, wait)`, while Linux has a function `kmalloc(size, flags)`. We abstracted these into a single `OSDEP_MALLOC(size)` function, which does what we really need each to do.

We created a lot of `OSDEP_`*x* macros to abstract away minor system-dependencies in our code, and we found that this approach works very well for small differences. Another common approach to the same problem is to use conditionals around blocks of code, and to provide an alternative for each system. Figure 1 shows an example of the code that results from this approach. For small differences, we believe that abstracting leads to more readable code and makes it harder for system-specific code to get out of sync.

Abstracting is also far more convenient for debugging than conditionals. The code in Figure 1a might expand to end up the same as the code in Figure 1b. But, when built with debugging enabled, `OSDEP_MALLOC()` actually gets defined as a call to our `malloc()` debugger code, and `OSDEP_RETURN_ERROR()` actually gets defined as a set of statements that tell us what line threw the error. Changing the code in this way is easy to do with macro abstractions, but would be painful to do with code surrounded by conditionals.

One of the serious dangers of conditionals is the temptation to make them either-or statements. For example, our earlier code contained conditionals that evaluated to one block on Linux systems and another block on non-Linux systems. This hard-codes a very dangerous assumption: that you won't be adding new ports to the mix that will make life less simple. We have been bitten a number of times by conditionals of this form when adding new ports, and we learned the hard way to be very careful with the preprocessor's `else` directive.

An important side note is that macros are strongly preferable to functions in kernel space. For applica-

a.

```
  if (!(pfkeyv2_socket = OSDEP_MALLOC(sizeof(struct pfkeyv2_socket))))
    OSDEP_RETURN_ERROR(ENOMEM);
```

b.

```
#if __bsdi__ || __NetBSD__ || __OpenBSD__ || __FreeBSD__
  if (!(pfkeyv2_socket = malloc(sizeof(struct pfkeyv2_socket), M_TEMP, M_DONTWAIT)))
    return ENOMEM;
#endif /* __bsdi__ || __NetBSD__ || __OpenBSD__ || __FreeBSD__ */
#if __linux__
  if (!(pfkeyv2_socket = kmalloc(sizeof(struct pfkeyv2_socket), GFP_ATOMIC)))
    return -ENOMEM;
#endif /* __linux */
```

Figure 1: Two Approaches to Minor Differences: (a) abstracting (b) conditionalizing

tion code, the overhead of a function call is a small price to pay for compatibility. In kernel space, performance and memory usage (both in terms of code size and stack use) is much more important, and so adding function calls that contain little code is probably a bad thing to do. In the more general sense, whether to put things in functions or to inline them as macros is still a judgment call the programmer has to make.

For larger differences, abstraction takes much different form. There, large functions or sets of functions, all surrounded by conditionals, is probably a reasonable approach. For example, a large part of our PF_KEY implementation is the interface between our messaging code and the system's socket layer. The different systems' socket layers required radically different interface functions and data structures to be provided. Since the organization and structure of what had to be done varied so much between the systems, we provided separate versions of these functions for Linux and for the BSDs, with more conditionals surrounding some of the differences among the BSD functions. However, the sockets interface code provides a uniform interface to the rest of our PF_KEY code, so the same messaging code can send up a message through the Linux sockets layer, the FreeBSD sockets layer, or the BSDI sockets layer, and the messages have essentially the same format.

Another large difference that needs abstraction is significant data structures. BSD systems use a chain of `struct mbuf`s to hold the contents of packets, while Linux systems use `struct sk_buff`s. These are very different structures, so abstraction of significant accesses to these structures won't work... at least, not without either making assumptions or serious performance degradation. In our PF_KEY code, we were relatively lucky in that we were able to simply define macro functions that copy blocks of data into and out of the systems' native buffers and were done.

In our IPsec code, however, we were not nearly so lucky – now we need to take a packet in the native buffer, operate on it, and return a result in the native buffer, and doing that by copies is not reasonable because it uses too much extra memory and costs too much for acceptable performance. So we defined an abstract buffer structure – the `struct nbuf` (see Section 3.3 for more details) – and defined "border functions" that take a native buffer and turn it into a `nbuf`, or take a `nbuf` and turn it into a native buffer. Those border functions are designed to use certain tricks and to check for useful cases that avoid copying the actual buffer contents in the common case, so the cost of using the abstract `nbuf` rather than the system-native structure is only that of the `nbuf` header itself and the cost of going through the border functions, both of which are small costs. The benefit is, in our opinion, huge. We now have a uniform, reasonable buffer that we can work with regardless of the system and a set of known properties of that buffer we can use to write simpler and faster code.

Which portability approach to take is basically a trade-off, and knowing which of the available options to take is basically a matter of experience and intuition. There is almost always more than one way that you can do it, and you can always go fix things later if you decide that you made the wrong choice.

2.3 Debugging

Each of the five systems that we support has different debugging capabilities. Many of them have the same facilities available (e.g., `kgdb`, `kdebug`, `ddb`, core dumps, display of trap information), but the reliability and net utility of those functions varies dramatically from system to system and among hardware platforms on the same system. I've never met an experienced kernel programmer who won't admit that all kernel debugging facilities are at best a mixed blessing: handy when they work, but they don't always work when you need them to. When your code has bugs that go trashing things in kernel space, it's not too hard for it to trash things that debugging facility needs (or the debugging facility itself, for that matter!).

There are several rather immortalized Linus Torvalds quotes about kernel debugging, but the summary of them is pretty simple and exactly agrees with our experience: The best approach to debugging kernel code is to read it, and read it carefully. Debuggers are a wonderful thing, but they tend to entice you into an interactive mode of programming where more time is spent trying to figure out if the code you have does the right thing than trying to figure out if the code you have is *right*. Especially when you're running on systems you just ported your code to and aren't as experienced with, reading your code and reading the system-native code you call is a useful thing to do when you're having trouble.

Though it's certainly a lot less convenient way to do things, we've found that the one relatively constant thing across the kernels we support is good old `printf()` (well, Linux calls it `printk()`, but a simple preprocessor `define` statement solves that problem). With only one exception (4.4BSD/sparc at high software priority levels), you can always use it to print data out to a console. By inserting `printf` statements in interesting places, you can binary search for the trouble spot and display the contents of variables that are interesting to the trouble code. This is a lot slower way to do things than attaching a debugger and single-stepping, but there are a lot fewer things that can go wrong and it seems to work on all systems. Using `printf()` also tends to change the execution properties of code being observed much less than kernel debuggers, which is important for certain classes of bugs. Code that starts working fine when under a debugger is not fun to debug.

Another set of tools that are commonly available and handy are the object tools. One technique we use all the time is to take the instruction pointer and stack contents from a trap, run `nm | sort -n` on the kernel binary to get the addresses of functions and data structures, and to figure out what function was executing, what functions called it, and what global data structures it was working with. If you built a kernel with debugging symbols, more detailed execution information can be gotten by running `gdb` on the image, listing the function you found was executing, and using the `info line` command to binary search for the actual line of code. Systems with the GNU toolchain have a command `addr2line` that does this for you, which is quite handy. Note that trap information can be misleading for certain types of bugs, so be suspicious if the information you get doesn't make sense. For example, we have found bugs where code trashes the stack frame and the function's return will cause execution to jump off into space; the trap address in that case can be all sorts of interesting values, none of which tell you where the bug is.

3 Detailed Discoveries

3.1 Differences Between BSDs

BSD/OS, FreeBSD, NetBSD, and OpenBSD are all derivatives of the 4.4BSD-Lite released from the University of California, Berkeley (most if not all of them have been updated to incorporate the patches in 4.4BSD-Lite2). In this paper, I don't have enough space to do justice to these systems' evolutions beyond this common ancestry, but I think it's important to describe some of the differences that affected our code. These are almost all local to the network stack.

In my opinion, BSD/OS has remained the closed to the shared ancestor, followed by OpenBSD and then NetBSD, with FreeBSD most aggressively changing from the common base. Many of the changes that the systems have made are not clearly good or bad, but are design trade-offs. The same set of changes is frequently considered evolution by some and devolution by others. From the point of view of portability, however, extraneous differences are bad because they require more abstractions or conditionals. In discussions with some of the systems' maintainers, this is not considered to be a problem, because portability of kernel code is not a priority. Hopefully, the maintainers of systems will reconsider this in the future.

One of the most mundane yet more prevalent changes the systems made is to make linked lists use the BSD `sys/queue.h` `TAILQ` macros rather than each having different implementations. Between the four BSD systems we support, there are three different ways this ended up getting done. For example, the interface address lists (`struct ifaddr`) are done in FreeBSD as TAILQs in with a field named `ifa_link`, NetBSD and OpenBSD as TAILQs with a field named `ifa_list`, and in BSD/OS as a normal linked list implemented explicitly. The main reason I mention this change is that it's so simple, everyone is basically doing the same thing, yet three different cases have to be handled. This happens often between the BSD systems (and between them and Linux, too, though not as often) – the same exact thing is done slightly differently by different systems. It would be really helpful if more effort were put into converging such things; this would make it a good bit easier to port code between kernels.

Another common change is that NetBSD and OpenBSD made protocol input or output functions use varargs rather than fixed parameters (as is still the case in BSD/OS and FreeBSD). This is good in that it allows certain I/O functions to have more parameters added without requiring every input or output function on the entire system to have its argument list adjusted to carry a dummy argument or the type system to be defeated using casts (an unfortunate side effect of the use of a single `struct protosw` for all protocol families). But this is bad in that it introduces a new opportunity for bugs. Arguments expected by the function aren't passed by a caller, so whatever happens to be on the stack becomes the parameter. Also, it is not always so easy to determine when a particular input or output function might not be called (the wonders of function pointers), and using a function pointer to call functions that expect different arguments in the same place creates a bad problem. NetBSD uses the `flags` argument to `ip_output()` to determine which of a few optional arguments are present; this approach might lead to a reasonable solution to the problems I found with using varargs.

There are also post-4.4BSD features that the systems have added, where each system did something different. A great example of this is the systems' choices for addressing TCP SYN flood attacks. BSD/OS implements a SYN cache and leaves the PCBs alone, NetBSD implements a SYN compressed state engine, PCB hash tables, and separate-case PCB lookups, OpenBSD implements SYN cookies and simpler PCB hash tables, and FreeBSD implements PCB hash tables and separate-case PCB lookups. Each change both `tcp_input`'s handling of new connections and the way PCBs lookups are done in significant and different ways, and each requires a special case.

Another interesting change is that NetBSD and FreeBSD pass a `struct proc` around inside the networking stack, from which socket privilege decisions are made, while the old way of doing things as in OpenBSD, BSD/OS is to make that decision when the socket is created and set the `SS_PRIV` flag. The appearance is that this change was made so that sockets lose their privileged status when the process that holds them loses that status; the `proc` that is passed around currently always seems to end up being set to `curproc`. This change has security implications, and was probably made to solve a security problem – but it may create other security problems as side-effects. Neither way appears to be clearly better. But, again, a lot of conditionals get created by this change to carry around this extra argument.

3.2 Between BSDs and Linux

Linux is very different than the BSDs. Frequently, either the same thing or a similar thing is done, and these differences aren't so bad to work with.

For example, there are a lot of things between BSD and Linux where almost exactly the same thing has a different name. Linux names its exact-bit types one way – e.g., `__u32` – and BSD names its exact-bit types another way – e.g., `u_int32_t`. These types do exactly the same thing. In this case, I have tried to convince the systems' maintainers (with limited success) to make definitions available in kernel space for the POSIX standard names of these types – e.g., `uint32_t` – which would help writers of portable kernel code avoid this particular problem. Another example of this is Linux's `struct iphdr` and BSD's `struct ip`, which have different field names, but both define exactly the same data structure. It would be helpful for portability of system maintainers were to agree either to converge on a single definition for these things or to provide a common definition along with a system-specific definition with another name. Until this happens, these differences can be worked around by using preprocessor `define` statements to do a search and replace.

There are also a lot of things between BSD and Linux that work similarly enough to be interchangeable, especially if you don't need all the functionality that the systems can provide. A great example of this is the difference between Linux's `kmalloc(size, flags)` and BSD's `malloc(size, type, waitflag)`. Both provide a more flexible version of the standard C `malloc(size)` function. By abstracting both into a single `OSDEP_MALLOC(size)` function, they can be used interchangeably on their respective systems.

Another example of this is how the systems handle "fast" critical sections by preventing higher priority interrupt-driven functions from running. Linux uses `save_flags(flags); cli()` and `restore_flags(flags)` for this, which actually turns off CPU interrupts, while BSD uses `s=splnet()` and `splx(s)` to provide this sort of exclusion for network code (other priority levels are used for other types of code), which turns off some software interrupts. For small sections of code which can't delay interrupts long enough to be a problem, these are reasonably equivalent (for longer blocks of code, arguably, neither should be used, and we should really use locks). By abstracting these into `OSDEP_CRITICALDCL`, `OSDEP_CRITICALSTART()`, and `OSDEP_CRITICALEND()`, again, these reasonably equivalent things can be used interchangeably on their respective systems.

Then there are a lot of things that are similar at a high level but not so interchangeable. An example of this is Linux's `sk_buff` and BSD's `mbuf`. We actually took two different approaches to this particular difference, each based on different requirements of different blocks of code. In our PF_KEY implementation, we needed to form messages in our own data structures and then generate native buffers to pass to the sockets layer, and we also need to take native buffers passed from the sockets layer and form messages in our own data structures. Also, performance is not critical in this code. So we created interchangeable higher-level functions that copied data into and out of the system-native buffers and performed certain specific operations on those buffers. Examples of these are `OSDEP_DATATOPACKET()`, `OSDEP_ZEROPACKET()`, and `OSDEP_FREEPACKET()`. In our IPsec implementation, we had to work with the buffers without copies, and that required a different approach.

3.3 nbufs

One of the really new things that we developed in the course of making our code portable was `struct nbuf`, which is a portable packet buffer structure. We set the design goals of the `nbuf` based on what we considered to be the best features of the Linux `sk_buff`:

- Packet data is contiguous in memory
- Payload data is copied into its final place and the headers are assembled around it in the buffer's "slack space," which helps avoid copies

And we also included what we considered to be the best features of the BSD `mbuf`:

- Small header size
- Few extraneous fields

We also imposed two more requirement, special to how this buffer will be used:

- Converting system-native buffers to `nbuf`s must be fast in the common case
- Converting back buffers that were so converted must be fast in the common case

Note that the second requirement is not "converting `nbuf`s to system-native buffers must be fast in the common case." This is an important optimization for reduced code complexity that we can make because all of the performance-sensitive paths in our IPsec code take a system-native buffer, convert it to a `nbuf`, work on that, and then convert it back into a system-native buffer. Since we never currently start with a `nbuf` and convert that to a system-native buffer, we don't need to that to be a fast operation. Future uses of the `nbuf` might need that, and we have ideas as to how to do that, but the current implementation does not support that as a fast operation.

The arrangement of the data buffer itself and the contents of the buffer are very similar to that of the Linux `sk_buff`. Both consist of a block of space, with a portion in the middle used for packet data and some

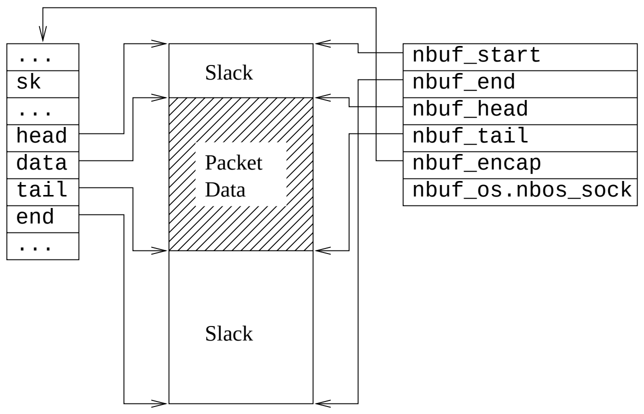

Figure 3: Encapsulation of a `sk_buff` in a `nbuf`

space before and after that left for expansion and/or padding. There are pointers to the start of the buffer (`nbuf_start`), to the end of the buffer (`nbuf_end`), to the head of the packet data (`nbuf_head`), and to the tail of the packet data (`nbuf_tail`).

There are also two special fields used by the border functions. The first, `nbuf_encap`, is a pointer to an "encapsulated" system-native buffer. In border cases that don't involve copies, a `nbuf` header is simply allocated and its fields point into the native buffer, in which case a pointer to the native buffer is kept to make it easy to "convert" back to the native buffer by simply updating the native buffer's fields, freeing the `nbuf` header, and returning the original buffer. The second special field, `nbuf_os`, is a structure that contains system-specific fields that must be copied in the "slow-path" cases where the system-native buffer data is copied to the `nbuf` and the original buffer is destroyed. When the `nbuf` is converted back, most fields in the system-native buffers can be filled in with reasonable default values without problems, but a few non-data fields must be filled in with the "right" original values. The `nbuf_os` structure allows us to ensure that we don't lose those values in the conversions.

Under Linux, conversion from a `sk_buff` to a `nbuf` is a fast path operation so long as enough "slack space" is available for the operations that are about to be performed on the `nbuf`. The Linux network stack already arranges for enough such space to be present in the `sk_buff`, and we extend that to include the overhead of the operations our code performs on the packets, so this fast path conversion should always happen in practice. Figure 3 shows what a `nbuf` looks like on Linux when a `sk_buff` has been encapsulated as part of a fast path conversion. There are some slight differences in the field names, but the fields in each structure are very similar, which makes the mapping between the two very easy.

Size	Typical Values	Typical Arrangement
0 .. MHLEN	0 .. 100	One mbuf
MHLEN+1 .. MCLMINSIZE-1	101 .. 208	Two mbufs or one cluster mbuf
MCLMINSIZE .. MCLBYTES	209 .. 2048	One cluster mbuf
MCLBYTES+1 .. ∞	2049 .. ∞	Multiple cluster mbufs

Figure 2: Typical BSD mbuf Arrangements for Various Sizes

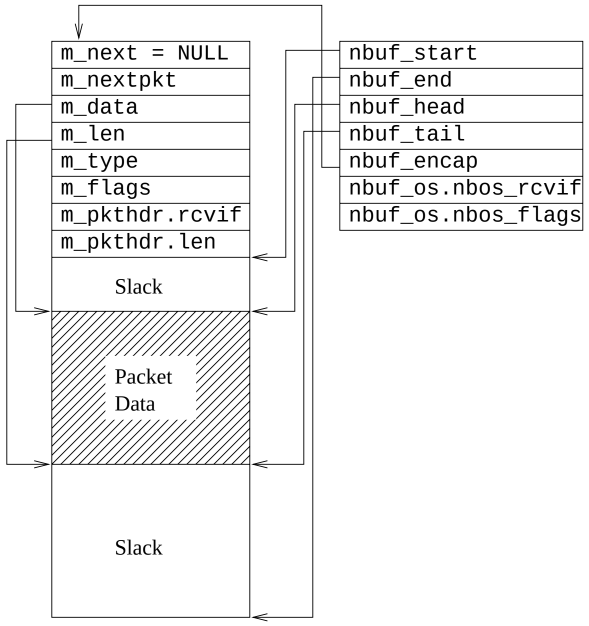

Figure 4: Encapsulation of a normal mbuf in a nbuf

Under BSD, conversion from a mbuf to a nbuf is a fast path operation so long as enough "slack space" is available and as long as all of the packet is contained in one mbuf. In 4.4BSD systems, the arrangement of the packet in buffers typically depends on the packet's size as shown in Figure 2. Typically seen IP packets tend to be either fairly small (about 44 bytes) or fairly big (about 552, 576, or 1500 bytes) [3]. The key observation we made is that, for the typical path MTU range of 576..1500 bytes, both typically seen categories are contained in one mbuf. Even allowing for the extra packet headers we add on output, we can make a fast path mapping between mbufs and nbufs for most actually seen packets. This is very important for good common-case performance on BSD systems.

Figure 4 shows how we encapsulate a normal mbuf in a nbuf. The nature of a normal mbuf packet header makes the locations nbuf_start and nbuf_end values fixed with respect to the start of the mbuf. The nbuf_head field is equivalent to the value of m_data, while the nbuf_tail field is equivalent to the value of m_data+m_len.

Figure 5 shows how we encapsulate a cluster mbuf in a nbuf. The nbuf_head and nbuf_tail fields relate to the m_data and m_len fields as with a normal mbuf. But now, the values of nbuf_start and nbuf_end are not fixed with respect to the mbuf; they are instead equivalent to m_ext.ext_buf and m_ext.ext_buf+m_ext.ext_size, respectively. Note that the data buffer attached to a cluster mbuf is always a size of MCLBYTES on the systems we care about, but we use the value in the field in case that changes.

nbufs have so far turned out to be very helpful in allowing us to make our IPsec implementation portable without sacrificing common-case performance. We have been looking at other possible uses for them, such as using them on systems different than both BSD and Linux and using them as a graceful transition mechanism between mbufs and a different buffer structure for the BSD network stack.

4 Results

4.1 Breakdown of Our Tree

Figures 6 and 7, and give some summary statistics about the portability methods we use in our source tree. These should give you a feel for what might be reasonable to expect out of porting kernel code. I interpret these statistics as providing cautious support for the idea of porting code between different kernels.

Figure 7 in particular deserves some extra explanation. Much of the work required for our IPv6 implementation is to "clean up" parts of the the BSD networking stack that were written to be IPv4 only (not an unreasonable assumption, but one that doesn't hold when we add IPv6). The result is that there are a lot of one line changes, which is why the ratio between changed lines and changed blocks is so close to one. Still, these are all system-specific changes. While the nature of the changes is similar among the different BSD systems, they must be done separately, by hand, and all kept synchronized. This is a lot of work, but the nature of the problem basically requires this approach. We tried to avoid maintaining these separately by using a common netinet tree, but that approach had its own problems. The lesson to be learned here is that there will always be a significant amount of system-

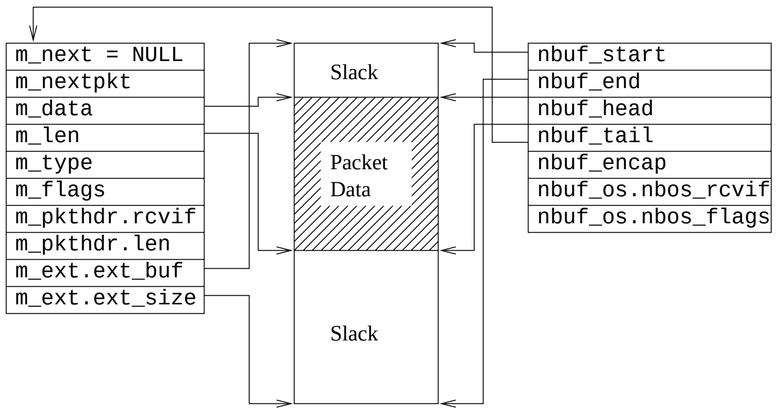

Figure 5: Encapsulation of a cluster **mbuf** in a **nbuf**

Category	Blocks	Lines	% Lines
Independent	N/A	14261	45.19
BSD only	N/A	13760	43.60
BSD/OS	28	102	0.32
NetBSD	30	71	0.22
OpenBSD	58	160	0.51
FreeBSD	156	931	2.95
Compound	371	1525	4.83
Linux[1]	44	735	2.33

Figure 6: Conditional code in shared trees

System	Blocks	Lines
BSD/OS[2]	1434	2410
FreeBSD	5388	5575
NetBSD	5080	5247
OpenBSD	4820	4951
Linux[1]	372	563

Figure 7: Changes to the systems' trees

specific code, and certain problems will naturally tend to require more of that.

The good news here is that we have a source tree total of 22270 lines (44.28%) of system-specific code, 13760 lines (27.36%) of BSD-specific code, and 14261 lines (28.36%) of system-independent code. Considering that this counts several copies of effectively the same thing, the nature of the IPv6 changes, and that most of the IPv6 code gets classified as BSD-specific rather than system-independent, this is pretty good. More than a quarter of our code is portable, and more than a quarter of our code at least runs on all of the BSDs we support.

The source tree totals penalize system-specific code more heavily when more systems are considered; if I simply omitted all support for a system, the percentage of system-independent and BSD-specific code would go up, and those percentages would give the illusion that more things are portable, even though removal of support for a system means the opposite is really true. Another way to look at these results that doesn't have this problem is to consider the breakdown of the code that goes into actual systems. Figure 8 shows the results of such a breakdown. On the BSD systems, the portable components make up about 40% each of the lines of code, with the system-specific components only being about 20%. On Linux, where we only support IPsec, the portable components make up 83.48% of the lines and the system-specific components make up 16.52%.

4.2 Conclusions

Based on these statistics, I conclude that, for code that can be reasonably ported, about one to two fifths of the source code will need to be written as system-specific code for each port and about three to four fifths of the source code can be made portable. That's still much better than half in common, and I believe that this provides cautious support for the idea that porting kernel code between significantly different systems can be a very practical and worthwhile thing to do.

Certain things are going to be inherently more or less system-specific. In our case, we had one thing that was not inherently very system specific (IPsec) and one thing that was more inherently system specific (IPv6). Even with the latter, we still achieved a good amount of portability. This is promising in that it suggests

[1] Our Linux support is a work in progress, for IPsec only, and does not include IPv6. The metrics presented for Linux are for example and aren't directly comparable to the other systems.

[2] Some of our code is already integrated into BSD/OS, which reduces the system-specific changes for that tree.

	Lines (%) of Code		
System	System-specific	BSD-specific	System-independent
BSD/OS[2]	4037 (12.59)	13760 (42.92)	14261 (44.48)
FreeBSD	8031 (22.28)	13760 (38.17)	14261 (39.56)
NetBSD	6843 (19.63)	13760 (39.47)	14261 (40.90)
OpenBSD	6505 (18.84)	13760 (39.85)	14261 (41.31)
Linux[1]	2823 (16.52)	N/A	14261 (83.48)

Figure 8: Breakdown of Code by System[3]

that a broad class of things could be practical to make portable.

The five systems that we support are different, yet they are still very similar. Proponents of some of these systems will try hard to deny this, especially in the case of the differences between BSD and Linux, but code doesn't lie. At least, not as much.

5 Acknowledgments

The NRL IPv6+IPsec distribution is the result of years of work from a lot of people. Other than myself, the implementation team includes or has included Randall Atkinson, Ken Chin, Daniel McDonald, Ronald Lee, Bao Phan, Chris Telfer, and Chris Winters. The software's evolution and a lot of our portability efforts were the combined result of everyone's effort, and they are each at least as deserving of credit as I am.

The most recent portability work, including our current source tree organization and `nbufs`, was done by myself, Ronald Lee, Chris Telfer, and Chris Winters.

I'd like to thank those members of the communities surrounding the systems we support who have helped us. In particular, David Borman, Alan Cox, Theo de Raadt, and Perry Metzger. If it weren't for their help, we wouldn't be able to currently support the systems we do.

And, of course, we all must not forget to thank the developers of the systems that we run on, especially for the free systems. It's a *lot* of work to develop and maintain an entire operating system. It's amazing that small groups of people can do this, frequently in their "spare time," and produce such high-quality results. If they didn't do this and make the source so easily available (if not free), my group at NRL, and many more like us, might not have systems to develop on, or we might not be able to give our results away freely.

Ronald Lee and Angelos Keromytis provided helpful feedback on earlier drafts of this paper.

The work described in this paper was done at the Center for High Assurance Computer Systems at the U.S. Naval Research Laboratory. This work was sponsored by the Information Technology Office, Defense Advanced Research Projects Agency (DARPA/ITO) as part of our Internet Security Technology project and by the Security Program Office (PMW-161), U.S. Space and Naval Warfare Systems Command (SPAWAR). I and my co-workers really appreciate their sponsorship of NRL's network security efforts and their continued support of IPsec development. Without that support, this paper, and this software, would not exist.

References

[1] Randall J. Atkinson, Ken E. Chin, Bao G. Phan, Daniel L. McDonald, and Craig Metz. Implementation of IPv6 in 4.4BSD. *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[2] D. L. McDonald, C. W. Metz, and B. G. Phan. PF_KEY Key Management API, Version 2, RFC 2367, July 1998.

[3] K. Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements From an Internet Backbone. *Proceedings of INET '98*, July 1998.

In addition to formal references, this work and this paper refers heavily to the source code for the five systems. For more information on each, go to:

BSD/OS	`http://www.bsdi.com`
FreeBSD	`http://www.freebsd.org`
NetBSD	`http://www.netbsd.org`
OpenBSD	`http://www.openbsd.org`
Linux	`http://www.linux.org`

For more information about the NRL IPv6+IPsec distribution, or to obtain the code, go to:
`http://www.ipv6.nrl.navy.mil`

[3] "Complex" system-specific blocks are counted as if they were included as system-specific blocks in all systems. This slightly under-represents the portable components.