Research Report: On the Feasibility of Retrofitting Operating Systems with Generated Protocol Parsers

Wayne Wang
Middlebury College
Middlebury, VT
Email: sw@middlebury.edu

Peter C. Johnson
Department of Computer Science
Middlebury College
Middlebury, VT
Email: pjohnson@middlebury.edu

***Abstract*—Parser generators show great promise in producing more secure input validation and processing routines. Operating system kernels are a particularly appealing place to deploy generated parsers due to their position at the periphery of a machine's attack surface, the power they wield, and their complexity. At the same time, kernels can also be byzantine and idiosyncratic. Before we attempt to create generated parsers for wide deployment into kernels, therefore, it is important to understand how much of the existing codebase those parsers will replace, what (if any) functionality beyond parsing the generated code will need to implement, and what kernel facilities the code must integrate with. To answer these questions, we analyzed three protocol implementations in each of three open-source kernels to understand their behavior and organization. We identified commonalities and differences, measured the quantity of code used for different purposes, identified when and how multiplexing decisions were made, and analyzed the order in which operations were performed. We intend this analysis to inform the creation of generated parsers, for a variety of protocols, that can be smoothly integrated into modern kernels.**

***Index Terms*—protocol stacks, generated parsers**

I. Introduction

Parser bugs are a common source of software vulnerabilities [2]. One approach to reduce the likelihood of such bugs is to *generate* parsers using programs instead of writing them by hand: such a *parser generator* takes as input a description of valid inputs and produces as output code that recognizes those inputs. If the generator is written to produce a parser that implements secure programming practices, all parsers it outputs will have the benefit of those practices. Additionally, if a new method or practice is developed, one need only update the generator and re-generate all the parsers to deploy a new layer of defense. Furthermore, it is often easier to verify (by hand) the correctness of the description of valid inputs than it is to confirm that a hand-written parser correctly implements a given protocol specification.

Many approaches to the task of parser generation have been developed over the years, resulting in tools that streamline the process of integrating the generated code into a software project [14, 22]. Such integration is much easier when a piece of software is designed with the intention of using a generated parser and, while there is much opportunity for improving security by encouraging authors of new software to use parser generators, there is also a great deal of existing software that would benefit from those generated parsers as well.

Operating systems, in particular, are an especially appealing target for generated parsers. By their very nature, this class of software implements parsers for a huge number of protocols and bugs in any of them can potentially compromise the entire system due to operating-system code running at a higher privilege level. While the idea of re-engineering kernels wholesale is appealing and has seen spectacular progress [13], these more-secure kernels have not yet made their way to the servers, desktops, laptops, phones, watches, cars, and other devices that people use on a daily basis. Therefore, to broaden the benefit of generated parsers, it behooves us to deploy them in the operating systems that *do* run on those devices.

It has previously been shown that generated parsers can be cleanly integrated into modern operating systems [12]; in that case, a parser for the USB protocol on FreeBSD. While this was a necessary stepping-stone on the path to wide adoption, it only solved the problem for a single protocol on a single operating system. We hypothesize that both other protocols and other operating systems will pose different problems. For example, different operating systems might have different mechanisms or patterns for managing memory or performing synchronization; some might have single-threaded protocol stacks and some might be multi-threaded; some protocols might induce significantly different software architectures; and so on.

At the same time, it is unclear how much code within kernel protocol stacks is devoted exclusively to parsing (as opposed to tasks such as memory management, logging, interfacing with hardware, etc.) and therefore what quantity of code could be replaced by generated parsers. Relatedly, to integrate with existing kernels, generated parsers will need to conform to their data structures and APIs, the extent of which is also unknown.

In this paper, we present the results of our survey of three protocol implementations (IP, USB, and SCSI) in each of three open-source kernels (FreeBSD, Illumos, and Linux), which we performed to gain preliminary insight into all of these questions. While this is not an exhaustive survey of all kernel protocol stack patterns, given the wide deployment of those protocols and operating systems, we believe it gives a representative view that can inform the design, implementation, and deployment of generated parsers.

We traced the code path traversed by a transmission unit (e.g., a packet in the IP module) from its ingress point to its egress point of the protocol implementation and we annotated each line of code along that critical path according to its purpose (e.g., parsing, memory management, logging). Using this data, we present answers to the following questions:

- What classes of tasks are performed within a protocol implementation?
- Are there similarities in the order in which these tasks are performed between protocols or operating systems?
- How much code is devoted to each category of task in each implementation within each operating system?
- What aspects of incoming data are validated by each implementation and at what point during the processing of the transmission unit are they validated?
- What kind of multiplexing or routing is performed as the transmission unit traverses the code path?
- What general kernel facilities are used (e.g., for memory management or synchronization) and are the same interfaces used across all protocols within a single operating system?

We hope that our results can be used to inform the design and implementation of parser generators that target deployed operating systems. That said, we fully appreciate that hardware is very messy, often requiring a great deal of special-case handling [4], and so it would be unrealistic to assume that humanity is on the verge of replacing huge swaths of operating-system internals with generated code. At the same time, however, it is not clear how extensive those replacements *could* be. It is the potential that this paper also seeks to help quantify.

We proceed by describing our process (Section II), presenting and analyzing the results we produced (Section III), before concluding (Section IV).

II. Process

To answer the questions enumerated in Section I, we performed a manual analysis of the source code of three protocol implementations in each of three operating system kernels. We identified where a unit of data entered the code and where it exited; we then categorized each line of code traversed according to its purpose. Below, we describe the reasoning behind our choices of operating systems and protocols, along with further detail on the analysis we performed, including descriptions of the categories we identified. In Section III, we present our results.

A. Choice of operating systems

In an effort to analyze a representative set of operating systems, we chose FreeBSD, Illumos, and Linux. Specifically, the following versions:

- FreeBSD 13.0-RELEASE, with no errata patches applied;
- Illumos revision from 04 Jun 2021 (`37fc8a1`)[1];
- Linux version 5.12.9.

[1]Illumos does not use formal version numbers.

Our intention behind these choices was to balance popularity with variety of implementation approaches. Linux is the most widely-deployed open-source operating system in the world, and thus deserves our attention. FreeBSD is the most widely-deployed open-source descendant of the BSD family of UNIX. While it does not have the usage numbers of either Linux or even FreeBSD, Illumos (the successor of OpenSolaris) is nonetheless the most accessible open-source descendant of the System V release of UNIX. Since all three come from different software geneologies (BSD and System V deviated in the late 1970s/early 1980s and Linux has always been developed outside the original, core UNIX community), we believe these three present a reasonable cross-section of modern operating system kernels.

We considered others such as Plan 9 and microkernels, but felt that their lack of widespread modern deployment made them less representative, despite potentially presenting significantly different software architectures.

B. Choice of protocols

As with our choice of operating system kernels, we wanted to analyze protocol implementations that were both widely used and varied in their details. To that end, we chose to examine implementations of version 4 of the Internet Protocol (IP), version 3.0 of the Universal Serial Bus (USB) protocol, and the Small Computer Systems Interface (SCSI) protocol. We believe these fulfill our needs for several reasons.

- All three of these protocols are extremely widely deployed: billions of devices implement them and all relevant operating systems support them. We are not aware of research that shows it, but we would not be surprised if these were the three most prevalent protocols, in terms of data transferred, in the world.
- The three protocols are used in different domains: IP for interprocess communication, USB (usually) for communication with peripheral devices, and SCSI for communication with storage devices. As a result, the code that implements each protocol would embody different goals and tradeoffs, live in different areas of each kernel source tree, and quite likely be developed and maintained by different groups of people, all potentially resulting in varied implementation approaches.
- Because of their various histories, the hardware involved to support each, and the other kernel subsystems they interact with, we expected that IP, USB, and SCSI would naturally induce a variety of software architectures inside the operating system.

Despite the expected variations hinted at above, we knew we would find some similarities. Because all of these protocols must receive data from somewhere, we knew that each protocol implementation would have an *ingress point*, where a unit of communication entered the code module in question. Likewise, because all of these protocols act on behalf of other processes, each implementation would have an *egress point*, where the unit of communication was passed on to the “higher layer” for further processing. We considered these two points

Protocol	Ingress Point	Egress Point
IP	hand-off from link layer (e.g., Ethernet)	hand-off to transport layer (e.g., TCP, UDP)
USB	bus controller interrupt handler	hand-off to device-specific driver
SCSI	HBA controller interrupt handler	hand-off to filesystem or block I/O layer

TABLE I
INGRESS AND EGRESS POINTS IDENTIFIED FOR EACH PROTOCOL ANALYZED.

Operating System	Driver	HBA Manufacturer
FreeBSD	`siis`	SiliconImage
Illumos	`pmcs`	PMC-Sierra
Linux	`aic79xx`	Adaptec

TABLE II
SCSI HBA DRIVER EXAMINED IN EACH OPERATING SYSTEM.

the bounds of our investigation; the ingress and egress points we used for each protocol are enumerated in Table I, and are elaborated on below.

1) Internet Protocol, Version 4 (IP): The Internet Protocol (IP) is responsible for routing data packets between interconnected networks and is thus endemic to modern life; version 4 [17] is the dominant variant. IP comprises one layer of the Internet protocol suite; it is bordered below by link layer protocols such as Ethernet and above by transport protocols such as UDP and TCP. This protocol arrangement means that transmission units arriving at a kernel will usually be processed by a link-layer component (often some combination of hardware and software) before being delivered to the IP component; and that, when it has completed its own work, the IP component will deliver the packet on to another component such as TCP. Therefore, we considered the ingress point for IP to be where it receives a packet from the underlying link layer, and the egress point to be where it delivers the packet to the upper, transport layer.

Two of the operating systems we examined, FreeBSD and Linux, implement their IP stacks according to specific design criteria that promote flexibility, composability, and security. In the case of FreeBSD, its NetGraph system [19] defines a notion of processing nodes, interfaces between them, and a mechanism to connect them. The idea is that each node implements self-contained functionality that is applied to a single packet (e.g., handling IP, handling ICMP, handling TCP) and these nodes can be connected together to create a processing graph. (In this context, a graph is more suitable than a linear chain due in large part to the multiplexing features of protocols such as IP.) Somewhat similarly, Linux's networking stack is tightly coupled with NetFilter [1], a collection of hooks that allow packets to be verified, manipulated, logged, queued, and so on, as they traverse the system.

We considered including TCP in our analysis, but we concluded that the similarity in structure between IP headers and TCP headers, as well as the fact that the code to handle both are likely developed together, meant that significant variations (for our purposes) were unlikely. Additionally, we expected that much of the processing at the TCP layer was likely to be related to the TCP state machine, and thus more in the realm of session generation [21, 16] than parser generation.

2) Universal Serial Bus (USB): The Universal Serial Bus protocol is, as its name applies, a protocol to support many different devices types sharing a single, high-speed serial bus [9]. Similar to IP, it is a packet-based protocol that encapsulates and transports device-specific data. Unlike IP, however, the USB specifications dictate functionality all the way down to electrical signals on the wire. Therefore, for our analysis to be thorough, we treated the bus-controller's interrupt handler as our ingress point. Likewise, since USB multiplexes the single transport stream between multiple devices and processes, we treated the point at which data is handed to a device-specific driver as our egress point.

Several versions of USB currently exist, many of which are still in active use; we examined the implementation of version 3.0—often referred to as XHCI: the eXtensible Host Controller Interface [6]—because it is the most modern. All versions of USB support different types of data transfers: control messages, isochronous transfers, and so on. For our analysis, we looked at the "bulk" data transfer code path because it is the most general-purpose. Where IP must handle unsolicited and/or unexpected packets that arrive, USB is a request/response protocol: the host sends a request and the device responds to it. As a result, we expect to find more stateful computation in this component than in IP implementations.

We treated the XHCI interrupt handler as our ingress point and the hand-off to the specific device driver as our egress point.

3) Small Computer Systems Interface (SCSI): Much like the Internet Protocol became the de-facto standard for allowing packets to traverse interconnected networks, the SCSI protocol has become the de-facto standard for sending commands to and receiving results from storage devices. Disk drives attached via USB and Serial ATA use the SCSI command set, as do more exotic interconnects like Fibre Channel, and access to remote storage devices can be implemented across networks using iSCSI.

Like USB, the SCSI standards dictate the behavior of the system at the wire-level; but unlike USB, these standards do not extend to the interface between the host computer and the bus controller, known as the "host bus adapter" (HBA). As a result, HBA manufacturers have defined their own interfaces which necessitate their own drivers. This is demonstrated by a cursory examination of any of the kernels we looked at, which contain a single implementation of, e.g., XHCI, which is manufacturer agnostic; whereas they all contain several different HBA driver implementations from different manufacturers. Given that even open-source driver code is often written and provided by the HBA manufacturer, and we wanted a variety, we chose the following HBA drivers, which are listed in Table II.

Somewhat like USB, the SCSI architecture separates the notion of devices from that of a bus. And while USB allows vendors to define their own device-level protocols, the SCSI standards define the protocol used by individual devices such as rotational disks, tape drives, and optical disks[2]. Thus, there exist many fewer device-level drivers in the SCSI world than in the USB world.

Also like USB, SCSI is a request/response protocol; therefore, also like USB, we expect to see more stateful computation here than in IP.

In contrast to the variety of HBA drivers we chose, we decided to examine only one device-level SCSI driver in each of the three operating systems; specifically, we chose the driver that works with SCSI disks (as opposed to tape drives or optical drives). This is because we believe it to be by far the most frequently used. In FreeBSD, this is the `da` driver; in Illumos and Linux, the drivers are both called `sd`.

The device and driver model just described, which is induced by the SCSI architecture itself, leads to a corresponding software architecture in which a transmission unit is read off the HBA (usually using DMA) by the HBA driver and then handed to the device driver for further processing. All three operating systems we examined defined a layering, with the HBA driver at the bottom and the device driver at the top, as well as an interface for the two layers to interact. FreeBSD's interface is the Common Access Method (CAM) [18, Chapter 12]; Illumos' is the Sun Common Systems Architecture (SCSA) [20, Chapters 17 & 18]; Linux uses an unnamed three-layer approach [5] though, upon inspection, the middle layer is just glue and does not seem to comprise a distinct software component with a well-defined boundary.

For our analysis, we treated the ingress point to be the HBA driver's interrupt handler and the egress point to be the call to the upper-level I/O module (e.g., filesystem, block I/O).

C. Execution Traces

Once we identified the ingress point in each component, we traced execution to the corresponding egress point. If a function was called, we analyzed its body if the function either lay along the main path to the egress point or if the function operated on the contents of the transmitted data in any way. If neither of these conditions were true, we treated the function call as a single line of code. (For example, we felt that diving into the details of logging or synchronization functions didn't provide much insight to our questions.)

When execution encountered control flow structures (e.g., `if`/`else`, `switch`/`case`), we followed the common case. Our reasoning was that, because this is the most well-traveled path, it would be the most representative. As specific examples: in XHCI and SCSI, we followed the code path that dealt with successful command completions; in IP, we followed the code path that did *not* involve IPSec. Likewise, we included checks for data malformation, but did not trace into the code that runs in the case of malformation being found. We encountered very few loops; those we did find fell into two categories: looping through enqueued transmission units to call a handler function on each (in which case we treated the loop as a single line of code that transitions to another function); and looping through fields within a transmission unit (e.g., IP options), in which case we treated the entire loop as parsing.

[2]It is worth noting that USB also defines the "USB mass storage" device *class* [8], which is a standard for interacting with any storage device that follows those rules. It is essentially SCSI wrapped within USB.

Category	Description
parsing	turning raw bits into named and typed values
multiplexing	control-flow decision based on contents of transmitted data
state management	getting/setting values external to packet
memory management	allocating and deallocating memory, DMA operations
hardware manipulation	reading and writing registers, checking bus state
synchronization	mutex locking and unlocking, condition variable waiting and signalling
queueing	enqueueing and dequeueing
diagnostic	logging, tracing, and statistics
assertion	sanity checking

TABLE III
CATEGORIES USED TO CLASSIFY LINES OF CODE IN THE PROTOCOL STACKS EXAMINED.

Next, we took these execution traces and categorized every line of code within according to its purpose. Single statements that spanned multiple lines received multiple annotations (our reasoning being that these are likely more complex operations and therefore worthy of more weight). We had an idea of what categories to expect before we began, but we kept open the possibility that we would find something unexpected. (We did not expect to find assertions as prevalent as they were, at least in Illumos.) The final set of categories we applied is shown in Table III.

If a single line of code seemed to fall under multiple categories, we labeled it with the topmost category in that table (that is, the higher the category in the table, the "more important" we considered it). In a few very rare cases, particularly relevant operations such as parsing and multiplexing were performed on the same line, in which case we added an empty line annotated with the second purpose.

Here we provide more detail of what operations comprise each category we identified.

a) Parsing: The task of applying meaning to meaningless data takes several forms; in the code we examined, it was usually assigning a typed pointer to untyped memory or extracting sequences of bits from larger values and saving the results in typed variables (note that this includes endianness manipulations). This category also includes validation of the transmitted data. For implementations that contained support for user-defined filtering and validation rules (e.g., NetFilter on Linux, pf [11] on FreeBSD), we marked the invocation of those mechanisms as "parsing" but did not descend into their internals.

b) Multiplexing: Since all the code we examined was written in the C programming language, multiplexing was accomplished using the `switch`/`case` construct, jump tables, and function pointers. Jump tables were usually hard-coded (e.g., in IP implementations, indexed by protocol number to identify the handler function for the payload). Function pointers were usually used to process a response to a previously-sent request, and the target of the function pointer was set when the request was originally issued.

c) State management: Computation based on data separate from the transmission unit traversing the protocol. This includes decisions such as "does the destination address in this IP packet match the address of a local interface?", which uses a value from the IP packet but the "correctness" of the value depends on the state of the particular system and not the definition of the protocol (which is what distinguishes this example from parsing or validation—more on this in Section IV).

d) Memory management: Operations that request or release memory buffers from the kernel; syncing DMA memory; memory fences; flushing page caches.

e) Hardware manipulation: Reading and writing of registers; checking the state of the controller or bus. Due to the nature of the protocols in question, the IP implementations did not contain any hardware manipulation, USB implementations contained minimal, and the HBA layer of the SCSI implementations contained a fair amount (we quantify this in Section III).

f) Synchronization: Some stacks are multithreaded and therefore use mutexes (locking and unlocking) and condition variables (waiting and signalling) to coordinate between threads. We did not observe any semaphores or other synchronization structures.

g) Queueing: The multithreaded stacks use queues to delegate processing of tasks, usually handlers for various stages of processing, in the form of callback functions. Additionally, XHCI uses a ring-buffer to manage outgoing requests and incoming responses; interaction with those structures fall into this category as well.

h) Diagnostic: This category includes logging messages (such as for `syslog`), execution-tracing machanisms (e.g., DTrace [3] and SystemTap [7] probe points), and maintaining statistics (e.g., number of dropped IP packets for `netstat`, number of I/O errors for `iostat`).

i) Assertion: Verifying that hardware is sane or consistent; verifying that input pointers are non-NULL; etc. Any assertions dealing with the content of transmitted data were marked as parsing.

To sum up, we identified three protocols and three operating systems (and thus a total of 9 code modules), we identified the ingress and egress points in each, we traced execution between them, and we categorized each line of code along that path. The next section presents the results of this effort.

III. Results

Having performed the categorization described in Section II, we processed the results in several ways. We counted the number of lines of code that we identified in each category; we identified where multiplexing was being performed and what data determined the decision made in each case; we produced colorized code diagrams to get a sense of the general organization of each implementation; and we identified the general facilities used by each. All of these points of analysis are relevant to the effort of producing a mechanism to generate code for a variety of protocols and that can be integrated with a variety of operating systems. We present and discuss each of those efforts below.

A. High-level categorization

Tables IV and V show the amount of code—number of lines and percentage of total lines, respectively—dedicated to each categorized purpose with each stack. Note that the rows are ordered the same as in Table III: "more important" categories are nearer the top. Recall also that this is *not* the total number of lines of code in these modules: it is the lines of code along the critical path that interact with the content of the protocol transmission unit. These tables are not intended to convey a deep understanding, but several points are immediately apparent.

First and foremost, in all three IP implementations, a huge portion of the code is performing parsing (which includes input validation). Had we further subdivided the code modules, we would see that the majority of the parsing is dealing with IP options. This makes sense given their complexity (and, not at all coincidentally, their reputation as a breeding ground for vulnerabilities—there is even an RFC wholly dedicated to filtering IP options because of how fragile they can be [10]). As skewed as the numbers are towards parsing in the IP implementations, both USB and SCSI require a non-trivial amount as well.

This leads us to conclude that the amount of code that could be replaced with generated parsers depends upon the particular protocol in question. In IP, with its complex structure of options and commensurately complex code to parse them, upwards of 85% of the code has the potential to be replaced. In contrast, both USB and SCSI are potentially less lucrative opportunities for replacement or retrofitting. (These conclusions are fairly consistent across operating systems, which corroborates the intuition that driver code complexity is more due to the complexity of the underlying protocol rather than the kernel enclosing it.)

Secondly, all three kernels perform a similar amount of multiplexing in each protocol implementation. This is reassuring: as a spot-check on our analysis and as a predictor of the kind of processing a generated protocol stack would need to perform. That said, it would be informative to understand the meaning of the discrepancies; we analyze that in Section III-B below.

Each protocol shows different discrepancies in terms of state management. Linux seems to perform relatively little in IP; Illumos seems to perform relatively little in USB; and Linux seems to perform quite a lot in SCSI. Here is where our analysis is a bit deceiving: the state management can happen within functions into which our analysis did not descend. Alternatively, if state management happens inline, such that our analysis catches it, we mark it as such even if it doesn't depend on or manipulate the contents of the data transmission. An enhancement to our work would be to develop a more finegrained understanding of what state is being modified and at what level of abstraction.

In terms of memory management, FreeBSD is particularly parsimonious: almost certainly as an optimization measure, it minimizes allocating and deallocating memory in all three protocol implementations.

It is unsurprising that all three SCSI implementations interact with the hardware because all three include an HBA driver. What *is* perhaps surprising is that all three USB implementations interact with the hardware so much. We believe this is a consequence of the XHCI specification, which requires that state be synchronized between OS and controller.

None of the IP implementations do much synchronization: packets are handled in a single-threaded execution environment. By contrast, all three operating systems use multithreading—with queues, mutexes, and condition variables—to handle both USB and SCSI. In part, this is certainly due to the asynchronous, request/response nature of those protocols.

The queueing row corroborates this, with one caveat pertaining to USB. To achieve its high speeds, USB 3.0 (XHCI) uses a ring buffer in DMA memory shared between the operating system and the bus controller, to which requests are queued and from which responses are dequeued. We marked operations interacting with this ring buffer as "queueing", but they do not fall under the banner of multithreading. (On the other hand, one could make the argument that they fall under the banner of *multiprocessing*, as they are used by the CPU to delegate a task to the XHCI controller.)

Illumos in particular really likes its assertions, whereas they are effectively absent in both FreeBSD and Linux. It would be interesting to know the underlying design philosophies and histories that led to this situtation.

Finally, it seems as if FreeBSD just uses a lot more code: for both IP and USB, the FreeBSD implementations are significantly larger. On the other hand, consider Table VI, which shows the number of functions touched in each implementation. (By "touched", we mean that we descended into them for our analysis: they either lay on the critical path between ingress and egress points or they manipulated the transmitted data.) It could be that the FreeBSD design philosophy encourages less abstraction in these components (and it could also be that their USB developers didn't get the memo).

All of these points work to inform decisions that must be made when designing generated protocol stacks: IP options are not to be underestimated; different kernels use different levels of abstraction; some protocols favor asynchronous, multithreaded, queueing environments and some do not; some protocols require significant interaction with the underlying controller hardware, some do not.

B. Multiplexing

All protocols we examined support multiplexing different communication streams over the same transport layer and underlying communication medium. As a result, each implementation performs some kind of multiplexing on the software side based on internal state and data within a transmission unit. We identified every point where such multiplexing occurs, recorded the determining factor in the branching decision, and the language construct used to implement each. Tables VII, VIII, and IX show the results for IP, USB, and SCSI, respectively. Rows are ordered according to the sequence in which the multiplexing decisions are made: earlier decisions are listed above later decisions. Empty cells indicate that the operating system in question did not perform multiplexing based on that row's criteria.

The information in these tables is valuable to our long-term goal of generating parsers that can be inserted into a variety of operating system protocol implementations. The data provides guidance for what kinds of decisions the generated code should probably make, the data it needs to make them, and mechanisms it might use to implement them.

We next elaborate on each domain.

1) IP: Unsurprisingly, all implementations perform multiplexing based on the protocol field of the IP header: this is what determines whether the IP packet contains an ICMP datagram, a UDP datagram, a TCP fragment, and so on—it is fundamental to the concept and functioning of the Internet Protocol. What is interesting is that both Linux and FreeBSD define jump tables enumerating all protocols that can be encapsulated within IP, whereas Illumos uses a `switch` statement. Further differentiating Illumos is that, near the beginning of its processing of a packet, it multiplexes based on the (type of) destination: local, loopback, multicast, broadcast, forwarding, etc. Linux and FreeBSD do not appear to have a similar decision point within the IP-handling code proper; we hypothesize that is it handled at the Netgraph (FreeBSD) and NetFilter (Linux) layers.

2) USB: In the USB realm, the methods used by Linux and Illumos were most similar and FreeBSD was the outlier (this discrepancy is corroborated by the graphical representations of the code presented in Section III-C). At the beginning of handling an XHCI interrupt, all three operating systems use a `switch` statement to route computation based on the type of the XHCI event being processed; and at the end, all three use a callback to pass the data to the device-specific driver that initiated the original request. In the middle, both Linux and Illumos multiplex based on the endpoint type (control, isochronous, and bulk); interestingly, Illumos uses a `switch` whereas Linux uses an `if`/`else`. This is likely because there are only three options to choose from.

As seen in Table VIII, FreeBSD is a bit strange: it uses two different callbacks, both set when the transfer is initialized, but invoked at different points in the processing sequence. One is called when the response is removed from the bus-specific queue of tasks to be processed; the other is called when processing the result of a single transfer. We believe this is the result of the multithreading model that FreeBSD chose to implement, and thus should be kept in mind when considering how to integrate with that operating system.

	IP			USB			SCSI		
	FreeBSD	Illumos	Linux	FreeBSD	Illumos	Linux	FreeBSD	Illumos	Linux
parsing	269	153	211	51	32	22	17	18	27
multiplexing	1	2	1	4	4	3	1	2	3
state	33	39	7	44	15	45	22	21	50
memory management	0	7	7	2	5	7	3	5	6
hardware manipulation	0	0	1	17	5	32	4	3	11
syncronization	2	0	0	16	14	1	6	12	1
queueing	0	1	0	48	6	8	9	8	3
diagnostics	8	8	1	39	11	6	7	9	19
assertion	3	6	0	0	9	0	3	28	0
total	316	216	228	221	101	124	72	106	120

TABLE IV
NUMBER OF LINES OF CODE DEDICATED TO IDENTIFIED PURPOSES WITHIN EACH PROTOCOL STACK.

	IP			USB			SCSI		
	FreeBSD	Illumos	Linux	FreeBSD	Illumos	Linux	FreeBSD	Illumos	Linux
parsing	85.1%	70.8%	92.5%	23.1%	31.7%	17.7%	23.6%	17.0%	22.5%
multiplexing	0.3%	0.9%	0.4%	1.8%	4.0%	2.4%	1.4%	1.9%	2.5%
state	10.4%	18.1%	3.1%	19.9%	14.9%	36.3%	30.6%	19.8%	41.7%
memory management	0.0%	3.2%	3.1%	0.9%	5.0%	5.6%	4.2%	4.7%	5.0%
hardware manipulation	0.0%	0.0%	0.4%	7.7%	5.0%	25.8%	5.6%	2.8%	9.2%
syncronization	0.6%	0.0%	0.0%	7.2%	13.9%	0.8%	8.3%	11.3%	0.8%
queueing	0.0%	0.5%	0.0%	21.7%	5.9%	6.5%	12.5%	7.5%	2.5%
diagnostics	2.5%	3.7%	0.4%	17.6%	10.9%	4.8%	9.7%	8.5%	15.8%
assertion	0.9%	2.8%	0.0%	0.0%	8.9%	0.0%	4.2%	26.4%	0.0%
total	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

TABLE V
PERCENTAGE OF LINES OF CODE DEDICATED TO IDENTIFIED PURPOSES WITHIN EACH PROTOCOL STACK.

	FreeBSD	Illumos	Linux
IP	2	9	11
USB	13	9	9
SCSI	6	9	14

TABLE VI
NUMBER OF FUNCTIONS TOUCHED ALONG CRITICAL PATH IN EACH PROTOCOL IMPLEMENTATION.

	FreeBSD	Illumos	Linux
destination class		callback	
ip.protocol	jump table	`switch`	jump table

TABLE VII
IN EACH IP STACK, FIELDS UPON WHICH MULTIPLEXING IS PERFORMED AND THE METHOD USED.

3) SCSI: All three operating systems use a callback to invoke the device-specific driver (we traced the execution of the disk driver, as discussed in Section II-B3). The HBA drivers we examined in Linux and Illumos both use a callback function to pass a command completion structure to the higher layer; FreeBSD does not. We believe this is because the drivers

	FreeBSD	Illumos	Linux
XHCI event type	`switch`	`switch`	`switch`
transfer	callback		
endpoint type		`switch`	`if/else`
endpoint	callback		
transfer	callback	`switch`, callback	callback

TABLE VIII
IN EACH USB STACK, FIELDS UPON WHICH MULTIPLEXING IS PERFORMED AND THE METHOD USED.

	FreeBSD	Illumos	Linux
OS request handler		callback	callback
device driver	callback	callback	callback
storage layer			callback

TABLE IX
IN EACH SCSI STACK, FIELDS UPON WHICH MULTIPLEXING IS PERFORMED AND THE METHOD USED.

we examined in Illumos (`pmcs`) and Linux (`aic79xx`) were written to be as OS-agnostic as possible: the callbacks in question allow the operating system to supply a function by which the completed command exits the HBA-specific code and enters the general SCSI code. This suggests that FreeBSD's `siis` driver may have been written specifically for FreeBSD.

Linux's `request` structure, which is passed into the SCSI layer to issue a request, contains a function pointer that is called by the SCSI layer to deliver the result. FreeBSD and Illumos instead call a function in the storage layer (in both cases called `biodone`), whose single parameter is the request, and which presumably performs the demultiplexing outside the SCSI layer.

C. Sequence of operations

In addition to the operations performed by each stack, as discussed previously in this section, we were interested in their order and shape. Did the implementations follow a similar sequence of operations, a similar pattern? Did the raw number of lines of code belie significant differences in the length of those lines (lots of shorter lines might suggest comparable complexity to fewer, longer lines)? To study this, we produced the diagrams shown in Figures 1, 2, and 3. Each line in the figures represents a single line of code; the color matches the colors in Tables IV and V; the length of each line matches the (relative) length of the corresponding line of code.

As with the counts, the first thing that jumps out about the IP diagrams is the preponderance of parsing: the largest chunk of blue in each represents handling of the IP options. All three IP implementations do most of their state management at the beginning. FreeBSD performs more parsing at the beginning of the IP processing sequence than either of the others.

In the USB diagrams, we see that FreeBSD and Linux both put more hardware interaction directly in the critical path, whereas Illumos seems to have abstracted away those details. All three implementations seem to sprinkle parsing operations throughout; this could be evidence of shotgun parsers [15]. Again FreeBSD has the most lines of code, but here we see that many of those lines are very short: this supports the hypothesis that the code is not actually more complex; it may just follow different formatting guidelines. We see that the multiplexing points fall at roughly the same intervals in each, which suggests a similar pattern—this is encouraging evidence towards the practicality of a generated implementation.

Finally, the SCSI diagrams show the most variation between the three implementations. This could be due to the age of implementations (the SCSI standard has been around for decades and driver implementations perhaps have not received as much attention as, e.g., IP stacks due to the security and performance sensitivity of the latter).

IV. Conclusion

In the preceding pages, we described our work in tracing execution through the critical path of three protocol implementations in each of three open-source operating system kernels. We categorized each line of code along these critical paths, identified the purpose of each, and analyzed the results. We believe the resulting understanding of common patterns, constructs, and operating system facilities used will be beneficial in efforts to create frameworks that can generate parsers for varied protocols that can be embedded in kernels. Based on our categorizations, we conclude that generated parsers could replace large amounts of hand-written code—upwards of 90% by line count in IP, 30% in USB, and 24% in SCSI—made all the more robust if the correctness of these parsers is proven.

Therefore, we conclude that the feasibility of replacing kernel protocol implementations with generated parsers depends greatly upon the protocol in question. The complexity of IP (specifically due to its support for options) makes it a particularly compelling target. Furthermore, referring to Figure 1, we see that the parsing portions of the IP stack are generally fairly self-contained, which indicates that it should be possible to just drop in a generated parser. USB and SCSI do not, unfortunately, feature as much self-contained code, which means that automatically generating a significant portion of those implementations will require that the generated code be responsible for more than just parsing. While intuition may have pointed to the same conclusion, it is still beneficial (and, indeed, necessary to produce a generated parser) to quantify the bounds of the parser's responsibilities.

Additionally, the data presented in Tables VII, VIII, and IX, which summarize the multiplexing performed in each of the three protocol stacks, inform the design of generated parsers. As they enumerate both the fields upon which multiplexing is performed and the programmatic construct used to perform each multiplexing operation, they serve as checklists to ensure generated parsers are mindful of the relevant protocol fields and to provide suggestions for implementation patterns.

Because the topic of generating parsers for these environments was on our mind throughout the exploration and analysis documented in this paper, several questions occurred to us that we feel need to be addressed before going too far down the path of parser generation for operating systems. The two most significant are these:

First, what portion of relevant protocols are ripe for generation? Are there important protocols that require some hand-written components (likely due to undecidability of the underlying input language)?

Second, what is the extent of the task of parsing? Certainly it should encompass validation that can be performed just by examining the unit in question (e.g., "is the checksum valid?", "does the value of the length field match the length of the datagram?"); but does it extend to questions that depend on other state (e.g., "does the destination address of this packet match the address of a local interface?"). The answer to this question will greatly affect the ambition level of parser-generation efforts.

Future work entails taking the results produced in this paper (the patterns, interfaces, and so on that we documented and analyzed) and applying it to the task of parser generation: defining the bounds of the generated parser's responsibility; the interfaces it will present and use; definition and implementantion of any necessary shim layers; specification of input languages; and parser generation itself.

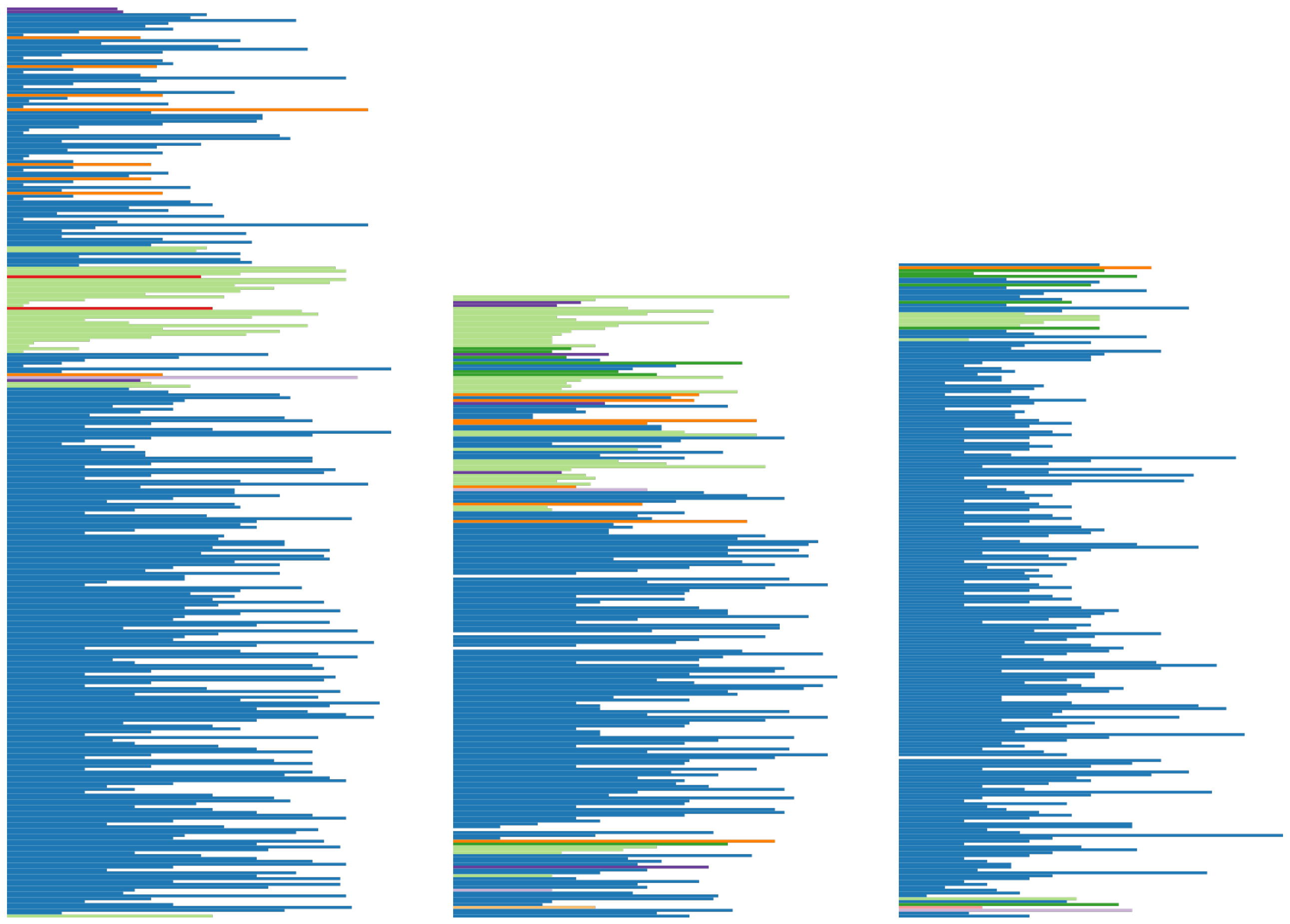
In the short term, we plan to enumerate all general kernel functions called from these protocol implementations; analyze their similarity both within a single operating system (e.g., do all protocol stacks use the same queueing interface?) and across operating systems; and propose an API for a universal shim that would allow a single generated parser to be integrated into a variety of protocol stacks within a variety of operating systems.

Acknowledgments

Our thanks to Christopher Andrews for his assistance in creating visualizations. This work was funded in part by Undergraduate Research Summer Funding from Middlebury College's Center for Teaching, Learning, and Research.

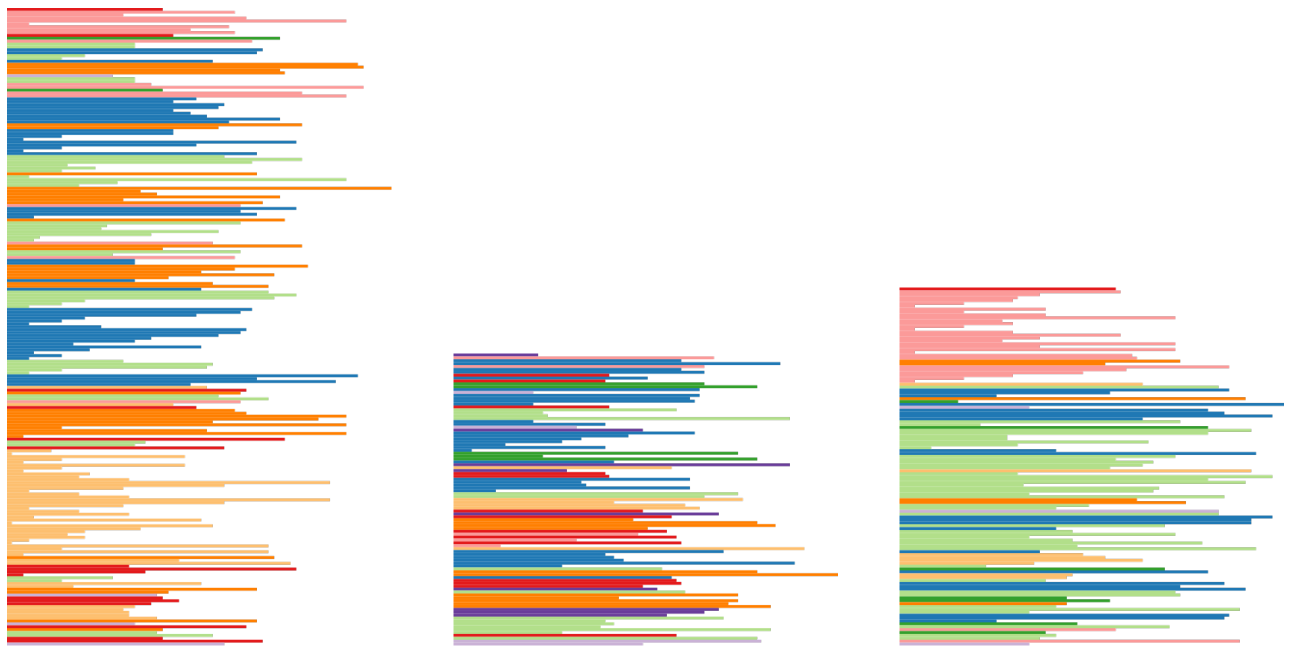
References

[1] Pablo Neira Ayuso. "Netfilter's connection tracking system". In: *USENIX ;login:* 31.3 (June 2006).



(a) FreeBSD (b) Illumos (c) Linux

Fig. 1. Categorized code diagram for IP. Colors correspond to categorizations shown in Tables IV and V; line width corresponds to the length of the individual lines of code.



(a) FreeBSD (b) Illumos (c) Linux

Fig. 2. Categorized code diagram for USB. Colors correspond to categorizations shown in Tables IV and V; line width corresponds to the length of the individual lines of code.

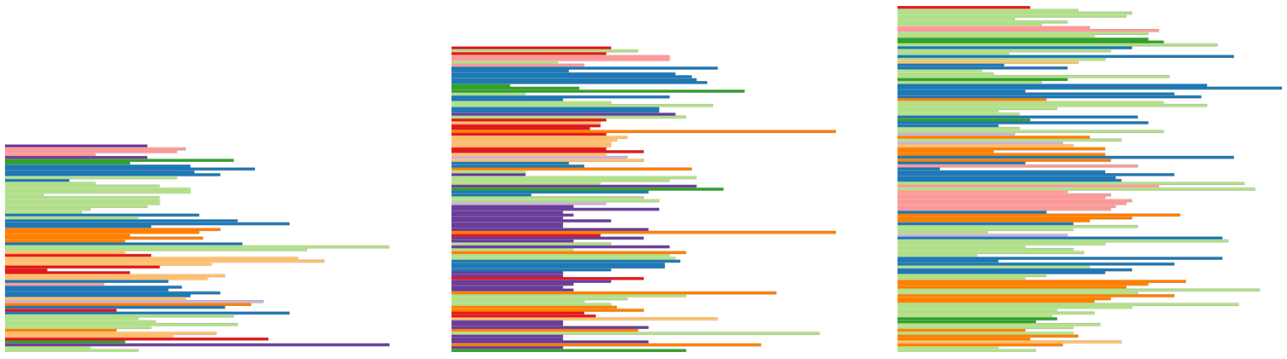

(a) FreeBSD (b) Illumos (c) Linux

Fig. 3. Categorized code diagram for SCSI. Colors correspond to categorizations shown in Tables IV and V; line with corresponds to the length of the individual lines of code.

[2] Sergey Bratus, Meredith L. Patterson, and Anna Shubina. "The Bugs We Have to Kill". In: *USENIX ;login:* 40.4 (Aug. 2015).

[3] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. "Dynamic instrumentation of production systems". In: *Proceedings of the USENIX 2004 Annual Technical Conference*. 2004, pp. 15–28.

[4] Vitaly Chipounov and George Candea. "Reverse Engineering of Binary Device Drivers with RevNIC". In: *Proceedings of the 5th European Conference on Computer Systems*. ACM, 2010, 167–180. DOI: 10.1145/1755913.1755932.

[5] The Linux kernel development community. *The Linux driver implementer's API guide*. 2021. URL: https://www.kernel.org/doc/html/latest/driver-api/index.html.

[6] Intel Corporation. *eXtensible Host Controller Interface for Universal Serial Bus (xHCI)*. Tech. rep. Version 1.2. May 2019.

[7] Frank Ch. Eigler et al. *Architecture of systemtap: a Linux trace/probe tool*. Tech. rep. Red Hat Software, IBM, and Intel Corporation, 2005.

[8] USB Implementers Forum. *Mass Storage Class Specification Overview 1.4*. Feb. 2010.

[9] USB Implementers Forum. *Universal Serial Bus Specification Version 2.0*. Apr. 2000.

[10] F. Gont, R. Atkinson, and C. Pignataro. *Internet Protocol*. RFC 7216. RFC Editor, Feb. 2014. URL: http://www.rfc-editor.org/rfc/rfc7126.txt.

[11] Daniel Hartmeier. "Design and Performance of the OpenBSD Stateful Packet Filter (pf)". In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference (FREENIX '02)*. 2002.

[12] Peter C. Johnson, Sergey Bratus, and Sean W. Smith. "Protecting Against Malicious Bits On the Wire: Automatically Generating a USB Protocol Parser for a Production Kernel". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 528–541. DOI: 10.1145/3134600.3134630.

[13] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 2009.

[14] Peter J. McCann and Satish Chandra. "Packet Types: Abstract Specification of Network Protocol Messages". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2000.

[15] Falcon Darkstar Momot et al. "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them". In: *Proceedings of the 2016 IEEE Cybersecurity Development Conference (SecDev 2016)*. 2016. DOI: 10.1109/SecDev.2016.019.

[16] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. "Protocol State Machines and Session Languages". In: *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW)*. 2015, pp. 110–119. DOI: 10.1109/SPW.2015.32.

[17] Jon Postel. *Internet Protocol*. RFC 791. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[18] The FreeBSD Documentation Project. *FreeBSD Architecture Handbook*. 2021. URL: https://docs.freebsd.org/en/books/arch-handbook/.

[19] Adrian Steinman. "Introduction to NETGRAPH on FreeBSD Systems". In: *Proceedings of the AsiaBSDCon 2012*. 2012.

[20] Inc. Sun Microsystems. *Writing Device Drivers*. Aug. 2008. URL: https://illumos.org/books/wdd.

[21] Kit Sum Tse and Peter C. Johnson. "A Framework for Validating Session Protocols". In: *Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW)*. 2017, pp. 110–119. DOI: 10.1109/SPW.2017.35.

[22] Yan Wang and Verónica Gaspes. "An Embedded Language for Programming Protocol Stacks in Embedded Systems". In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*. 2011.