In modern computing, even programs that seem relatively simple at the application level are rarely trivial in the eyes of the Linux Kernel. As the Kernel is the intermediary between hardware and the Operating System, Kernel space is often frequented by system calls from the OS to perform some physical operation at the physical level. In Project 1, a new TCP option, TCP Repeat is implemented which requires many detailed design decisions to properly execute in kernel space. The team started by determining the relative overhead our additional option would require. It was necessary during the three-way handshake for a new option flag to be set and checked both server-side and client-side to ensure both mediums supported the repeat options. Next, a methodology for actually transmitting repeat packets needed to be added. Finally, a system for tracking had to be addressed to ensure combined delivery to the application layer, ack packets, and allowing for checks to determine the proper packet for retransmission in cases of packet loss.

The most logical place to start was with the three-way handshake, as it is the beginning of the connection. To handle the SYN, we had to add a constant to bitwise compare with the options field of “struct tcp_out_options” and did the bitwise or during “tcp_syn_options()” within “tcp_output.c”. We also added an 8 bit field to this struct to store the i, n, and mode for the option (later, we made a second set of the option flag and field to handle the acks). We played with the idea of only setting this in the SYN only when told to do so during the connect system call, but decided that this would be too much extra work, and on the receiving end unsupported options on the SYN are supposed to simply be ignored. With the options set, they then had to be added when actually building the header, in “tcp_options_write()”.

With a working SYN, we then moved to tcp_input.c to parse the SYN. “tcp_parse_options” was the clear place to do this, and just like setting up the option for output, we added to the existing struct that was clearly for options, in this case “struct

tcp_options_received". Since we made a fresh field rather than working with bitwise operators like we had before (both times done to match existing code), we had to make sure this field was cleared, which was as easy as adding it to the function "tcp_clear_options()" right next to the struct declaration. Moving the detection of the option to the SYNACK was the first big hurdle. When we looked in "tcp_synack_options()" in tcp_output.c, most of the options are based on values in a request sock, so we had to add a field in the request sock (specifically in "struct inet_request_sock"). Once we had a field, we had to set it, which was handled in "tcp_openreq_init()" in tcp_input.c.

Now that we had a working three-way handshake, we could begin the core of the implementation. First, we threw together a system call that was really just a copy of "sendto()" that was stripped down to only support the logic needed for "send()" and had the ability to take the new argument requirements. With the beginning of a system call in place, we moved on to the second big hurdle, finding a time to actually make the packet repeat without duplicating the memory. After a close examination of the call stack, we determined that the repeat option and count should join the msghdr struct and would then make it all of the way to when the data is turned into packets in "tcp_sendmsg_locked()" in tcp.c. We were concerned about making sure the value was correctly initialized, and due to the large number of files it was referenced in this quickly became a difficult challenge for us. The best solution we could conceive was in "include/linux/socket.h:54" where we chose to modify the struct msghdr, doubling the size of msg_flags member from a u32 to a u64 in order to accomodate the count for TCP Repeat. Whenever we needed to read or write it, we just did a bit shift by 32. Since this field should be zeroed out to start whenever the struct is used, if there is a value in the new upper half we know it is ours. While this got the repeat count relatively deep in the sending call stack, we still wanted it deeper. Immediately after the packet was filled with data, we changed the sequence

numbers to cover the whole range that this repeat call would use. This meant both we could both that we could detect this packet later and that we could have only this copy stored for retransmission, using the TCP Repeat acks to figure out which repeats needed retransmission.

The next point at which we decided the TCP Repeat packets would need special handling is in "__tcp_transmit_skb()" in tcp_output.c. Since this is where outbound packets get their headers built, we needed to check if the current packet was a repeat packet and if so build the header with the experimental option included. Utilizing the previously modified msghdr in "include/linux/socket.h:54", we performed a check to see if the sequence number was more than twice that of a standard packet. If this was the case, we knew that the packet was a repeat packet and initialized a u8 repeat_i to keep track of the current packet number local to the function. If it was the first transmission, we would also initiate the repeat_out structure for handling outbound acknowledgements. __tcp_transmit_skb() then gets called recursively to handle all n iterations for TCP Repeat. The control buffer is copied and the skb is cloned n times, as determined by repeat_out.n. Finally, we added functionality using clone_it to clear the control buffer only when __tcp_transmit_skb() is given a zero or one value, as these were the only two cases we found to exist in the original code. We did this to ensure that we could know with certainty that we were handling a repeat packet, as opposed to running cloneit each instance initially.

Once packets were being repeated with the options in the header, we moved on to tackling ACKs. Our initial idea for tackling this was far too complicated. We thought up a way to allocate copies of a struct from an array in each socket, which would then be added to two separate lists based on the direction the repeat action was going. Halfway through implementation we realized that this was far too complicated, as the repeat option does not specify which set of repeat packets it is for, so there is no good way to tell the difference

between two different sets of repeat packets. We then simplified the struct and directly placed two in each tcp_sock, repeat_in and repeat_out. The final version of “struct tcp_repeat_ack_progress” contains the start and end sequence numbers, 3 bits each for i and n, and an last_ack field. The last_ack field is used to easily tell if the kernel can be done worrying about TCP Repeats for now and is thus ready for the next repeat set. It is set to one in the TCP_SYN_RECV case of “tcp_rcv_state_process()” in tcp_input.c on the listener side and “tcp_syn_options()” in tcp_output.c.

When an inbound TCP repeat option is parsed on a packet without the SYN flag, it follows one of two paths. The first path is for when the option is about inbound data, and the repeat_in copy of the struct is updated. If i is one, the struct is updated with all of the new values. Otherwise, the i stored is only incremented if the inbound i is one higher than the last recorded i. “tcp_established_options()” relies on these values and adds TCP Repeat’s ACK option on every outbound packet until it sends one where i and n are equal, at which point least_ack is set back to one and it will be skipped until the next set of repeat packets. For the second path, when the option is about an ack, the repeat_out copy of the updated with the incoming i (it was initialized for this repeat set in “tcp_transmit_skb()” before the first outbound packet was sent). If the incoming i is equal to n, last_ack is set to one to indicate that the kernel is ready for the next iteration and no longer has to worry about this one.

Since we decide to put these values in the tcp_sock struct, we had to add a way for this struc to be accessed from inside “tcp_parse_options()”. Our solution was to add a wrapper function with the original name, and rename the function to “__tcp_parse_options()”. This allowed us to add an additional argument for a tcp_sock pointer, which is set to NULL when called through the wrapper. We then changed the most likely point for it to be called

(“tcp_fast_parse_options()”) to actually pass the pointer to the tcp_sock, giving us access to these fields. This concludes our design decisions and discussion of our implementation.

When testing our code we only were able to test the pure case of the kernel interacting with itself. In the process of development, one of the team member had to increase the virtual disk size of their virtual machine, which caused many unintended and unforeseen issues that ultimately ended with the installation of an entirely new linux virtual machine. As a result of this and also with the time we had left, we were only able to verify the functionality of our kernel on one machine with a Wireshark packet capture of the machine’s localhost. We wrote a test script, syscall_test.c, which connects to a netcat port in listening mode and send “test\n\0” to the server repeated 5 times. Our wireshark packet capture verified that our changes to the kernel behave as intended and that our kernel handles TCP Repeat to the specifications provided. Over the course of this project, we learned a great deal about kernel development and also how difficult introducing and implementing new features can be. Overall, while we faced many challenges, we were able to find our way through and successfully implement the option.