Riley Rettig

CS 341 Operating Systems Final Project
TracerHUB

Introduction

For my project I wanted to focus on learning new tools as well as the basics of the interactions between operating systems and network cards. My project is a mix of implementation, measurement, and exploration that aims to find a solution for system tracing with minimum overhead. What I have done so far has just been the framework for a sort HUD or dashboard which, with more development, I hope would look a little like wireshark or a task manager.

This sort of tracing is important in the real world, since it can help define where we should be devoting our time with research advancements and building systems (such as recognizing something like an IO bottleneck before improving CPU speed), but can also help us determine what features in a system may be possible and for what values these features should have, such as maximum CPU utilization, fewest IO operations, etc. If we can do reliable tracing, we can recognize anomalies, which helps both in debugging as well as identify potential security threats.

Originally I wanted to focus a little more on networks, since networks in themselves are fascinating systems and there is a lot of overlap in considerations when designing a network and designing an operating system, which stems mostly from the last that there are limited resources that must be shared. For networks this is between users and for operating systems this is between processes. I thought that looking at the network card would be interesting, since it is just another form of I/O like the others we have been talking about in class, like the disk. A network card, like any other form of I/O either has to interrupt the OS to get things done in kernel space, or the OS has to utilize polling to “check in'' on the network device driver. I decided to look into using the Berkeley Packet Filter, which was originally designed in the 90s to analyze network traffic and filter incoming network packets[3]. As I explored the Berkeley Filter Packet, and what the extended

version has become today, I learned that there are so many different aspects of a system that can be tracked using BPF. My goal in this project was to learn how to use these new tools to track system calls in a more efficient and powerful way than the built-in linux tools as well as explain how they work to do this.

Design

The tool I used for my project is called bcc[1], which gives us an abstraction from eBPF, which is itself, an abstraction from BPF. bpf() is a system call which can be called using BPF byte code that is similar to assembly language. Luckily lots of smart people have abstracted that so that we can write both user code and kernel code simply using python and a restricted version of C. Under the hood, this C code will be running in kernel space without using system calls as a sort of liaison between user and hardware. The only system call used will be that bpf() call. This is great because it means we can directly observe the OS, but there must be security measures in place. BPF files (all the way down to the byte code) have a restricted program size, they can only jump forwards, all null pointers must be checked that they are not null, so as not to dereference them, and only bounded loops are allowed. This prevents unsafe memory accesses as well as denial of service attacks.

Implementation

Bcc provides many tools and examples, which were fun to play with and see what the system I was working on was doing, but I ultimately wanted to build a tool of my own to count the number of IO operations that were occurring and report it at a set interval. For this I had to utilize maps, and in particular, BPF_HASH. Part of what makes bcc so interesting is that it can listen to system calls/events and then make updates dynamically in memory. In my hello world script, I used bpf_trace_printk() to print to the trace_pipe that bpf uses. But this pipe is globally shared, so if I wanted to trace multiple things at a time they would have clashing output. Instead, I use BPF_HASH which, as you could assume, is a hashmap of key, value pairs. This means in kernel space I could save values to/update the value of a counter in memory, then in the python user space I could just access this hashmap to get the value at any time I'd like.

Evaluation

For evaluation, I used a tool called stress-ng, which can stress test a computer system in a controlled way, allowing you to pick which components are stressed. I have set up a few shell scripts running these stress-ng commands and then used a python script to measure the end-to-end time of these calls both with and without my tracing script turned on. Surprisingly, there was very little, and in some cases negative overhead, which I do not completely understand. For example, I ran a stress-ng command 100 times that included two IO stressors and 5000 operations, 50 with the tracer on and 50 with the tracer off. The average without the tracer was 9.26s and with the tracer was 8.80s. I see this as either my testing script is poorly designed, stress-ng is too variable/not a useful tool for this, or that my IO tracer just has a very low overhead since it is only tracing a single event.

Conclusions

I find bcc to be a really fascinating tool and I've really enjoyed learning how it links user and kernel space, even if it means my project is fairly slim on the "implementation" side of things. I am glad there are people out there who are enthusiastic about these low-overhead observability tools and are willing to communicate it, such as Brendan Gregg, who without his commit messages and tutorials [2], I would have been completely lost. Tracing does seem to be a really useful tool, and as systems become larger and more complicated with emerging technology, I think there will be less people who really know how all of the components work and tracing will help us identify issues as they arise. In particular, I would be interested in learning more about the possible security uses of tracing. Right now BPF is all about observation, and can't prevent attacks, but I'm thinking for some exploits that may run under the radar for a while, this may be a useful way to recognize suspicious activity.

References

[1] GitHub. “Iovisor/Bcc.” Accessed December 18, 2020. https://github.com/iovisor/bcc.

[2] “Learn EBPF Tracing: Tutorial and Examples.” Accessed December 18, 2020. http://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html.

[3] McCanne, Steven, and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture.” In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, 2. USENIX’93. USA: USENIX Association, 1993.