COMP9242 - Paper 2

z. . . . . .

November 9, 2014

1 Summary

This paper presents a new implementation of a network operating system designed for resilience of the controller and containment of individual applications.

The primary motivation behind this work is to safely support applications that may be malicious, buggy or use excessive resources. This is a goal that is not met by currently existing network operating systems, as demonstrated via a survey of existing systems and examples of programs which cause them to fail, where a single buggy application would also cause the network controller to crash, for faults as basic as the application exiting or allocating large amounts of memory.

As is common in most operating systems, handling of untrusted applications is achieved via introducing process context separation, resource utilization monitoring, and a permissions/capability structure. In Rosemary's case, this is implemented via a microkernel-inspired separation of functionality.

Namely, network applications are separate processes, each with their own copy of the required libraries, that communicate with the kernel and network services via an IPC mechanism. And secondly, network kernel service modules are also separated into running processes, allowing for a single service to fail and be restarted with no impact on other services - or for a single service that isn't required to not run at all.

The Rosemary implementation is tested to show that the implemented system successfully survives basic faults in user level applications, and gives some limited performance comparisons to existing NOS systems.

2 Pros

The abstraction of a NOS should be minimal, much like a microkernel, and the paper frequently states this.

The network services provided by rosemary are separated into different processes, which implements the ability that if a single service crashes, other services remain functioning, and the crashed service can then be restarted.

Implemented applications can be run as either a kernel level application or a user level application within the NOS with no alterations.

Vulnerabilities and flaws in other network operating systems are clearly demonstrated and explained, and similarly Rosemary clearly solves some of those flaws.

3 Cons

A lot of the paper discusses ideas that have been around for decades - context separation, resource management, access control etc. are not new ideas, nor is their implementation novel. That said, the paper does not read as though the authors claim to have invented this - the inspiration is clear. Similarly, the security flaws observed in NOS applications are extremely basic, and likely not hugely challenging to fix in existing NOS products.

A major flaw of the security model here is that the network operating system is still a user-level process running within a host kernel (generally Linux), and hence every child process is running at the same privilege level - any actively malicious network app could easily bypass anything implemented in this paper.

The paper repeats itself in several places - such as specifying in section 3.1 for both points 2 and 4 that a NOS needs to be minimal. The paper could be considerably shorter without this repetition - would there be any motivation to have wanted the paper to be 12 pages instead of 8?

4 Criticisms

To improve performance, Rosemary pins processes to adjacent CPU cores to match communication between pipelined stages. It would have been good to see some discussion of the potential issues with doing this - such as, say, when there are more threads than cores and it results in being unable to concurrently schedule two very active applications. They note later in the paper that the performance difference between Rosemary-kernel and Beacon could be due to this, but then dismiss it immediately.

The resource management is implemented as a policy - namely a static table (with an unspecified implementation.) Implemented in this way, it has the potential to lead to inefficiency - for example, a 1G cap on network usage for a single app with a 10G connection - and does not allow for anything more nuanced.

They mention formal verification as an alternative option for trusted applications, which is a good option where possible. However, the implementation of Rosemary is 20k LOC - a larger code base than several microkernels, and thus unlikely to be verified itself.

They state that if requested services from Rosemary to a network application were provided only via IPC, there would be considerable communication overhead. However, their alternate strategy is unclear - applications are instead provided libraries to communicate over the network. How does this provide the services - network I/O is not noticeably

cheaper than IPC - and presumably the data is local? This is a confusing paragraph. Later in the document, it states that all actual network packets are passed to applications through IPC, further confusing the issue.

Despite using a microkernel as the inspiration for their design, there is little discussion on the actual IPC mechanism - a domain socket - and the performance impact and trade-offs inherent in this.

Benchmarking

In testing each NOS, they state that they have optimized each according to its supplied recommendations, which is a good sign. It'd be nice to see further attempts at optimization, and slightly more detail as to the exact libraries and optimizations used. It would also be nice to see anything else that is not flows per second measured - such as latency!

No standard deviations or multiple test runs are ever mentioned. Notably the Figures 21 and 22 both use the 10G interface, and therefore the 8 thread/64 switch data should be identical, however it isn't. The K-App result is clearly around the 8.5M value in figure 21, while in figure 22 it's clearly around 9.75M - showing that there is most likely significant variation between different tests!

The test environment used on the NOS host has two 8 core CPU's - that is, a total of 16 cores. Interestingly in their benchmark results, they stop at 16 threads - a data curve that could easily be hiding deficiencies with more than one thread per core. Furthermore they state that Floodlight performance saturates after 15 threads based on a single data point!

The rosemary user level application shows extremely consistent performance - essentially flat - on all of the tests, with the single exception of a performance jump going from one thread to two, yet there is no discussion of this in the paper! It's clear that the situation here is that there is a bottleneck in communication between the kernel and user level threads, but no attempt is made in the paper to address this.

The authors seem to focus on the K-app benchmarks in this section, which is undesirable - the paper should be focusing on the performance of the user level apps that implement all of the features the authors have discussed. The kernel app benchmarks are primarily interesting in comparison to that of user level - to observe the costs of the new security features - but this discussion is lacking. The only mention is that they lose $20 - 30\%$ in 10G but 0% in 1G - an observation which is itself flawed. As is clearly visible in the graph, the network is saturated and therefore everything running is I/O bound. It is highly likely that if CPU usage was measured, there would be a noticeable difference - network saturation does not mean that there is no performance impact!

Finally there is no real analysis of the kernel app performance drop with additional switches - showing from 16 switches - a decrease not shown by any of the other NOS, and thus notable.