Rosemary: A Robust, Secure, and High-Performance Network Operating System

Seungwon Shin[*]
claude@kaist.ac.kr

Yongoo Song[†]
yongjoo.song@atto-research.com

Taekyung Lee[†]
taekyung.lee@atto-research.com

Sangho Lee[†]
sangho.lee@atto-research.com

Jaewoong Chung[†]
jaewoong.chung@atto-research.com

Phillip Porras[††]
porras@csl.sri.com

Vinod Yegneswaran[††]
vinod@csl.sri.com

Jiseong Noh[*]
jiseong.noh@kaist.ac.kr

Brent Byunghoon Kang[*]
brentkang@kaist.ac.kr

[*]KAIST [†]Atto Research Korea [††]SRI International

ABSTRACT

Within the hierarchy of the Software Defined Network (SDN) network stack, the control layer operates as the critical middleware facilitator of interactions between the data plane and the network applications, which govern flow routing decisions. In the *OpenFlow* implementation of the SDN model, the control layer, commonly referred to as a network operating system (NOS), has been realized by a range of competing implementations that offer various performance and functionality advantages: Floodlight [11], POX [30], NOX [14], and ONIX [18]. In this paper we focus on the question of control layer *resilience*, when rapidly developed prototype network applications go awry, or third-party network applications incorporate unexpected vulnerabilities, fatal instabilities, or even malicious logic. We demonstrate how simple and common failures in a network application may lead to loss of the control layer, and in effect, loss of network control.

To address these concerns we present the ROSEMARY controller, which implements a network application containment and resilience strategy based around the notion of spawning applications independently within a *micro-NOS*. ROSEMARY distinguishes itself by its blend of process containment, resource utilization monitoring, and an application permission structure, all designed to prevent common failures of network applications from halting operation of the SDN Stack. We present our design and implementation of ROSEMARY, along with an extensive evaluation of its performance relative to several of the mostly well-known and widely used controllers. Rather than imposing significant performance costs, we find that with the integration of two optimization features, ROSEMARY offers a competitive performance advantage over the majority of other controllers.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Network Operating Systems

CCS'14, November 3, 2014, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660353.

Keywords

Software-Defined Network (SDN); OpenFlow; Controller Robustness

1. INTRODUCTION

At the center of the growing emergence of the SDN paradigm is the notion of giving control of network flow routing decisions to a globally intelligent view, hosted above the data plane, that is able to coordinate across network components. SDNs enable this intelligence to be written in software, as *network applications*, using open APIs that better facilitate agile development and perhaps faster network innovations.

The OpenFlow stack is an embodiment of this notion. It offers a dramatic shift from the proprietary closed control layer of traditional network switches, to one in which the control layer exports both the API and the data plane abstractions necessary to facilitate a wide range of network applications. The introduction of the term *network operating system* (NOS) was born from the recognition of how the OpenFlow control layer provides network application developers with programming abstractions necessary to control the network data-plane hardware. The control layer provides interface and abstraction in a role analogous to that of operating systems, which provide software developers an appropriate abstraction for interacting with a host computer. Here, we will follow and extend this analogy, and will use the term network operating systems (NOSs) and OpenFlow controllers interchangeably.

Unfortunately, the myriad of parallel efforts to design NOSs (e.g., POX [30], NOX [14], Beacon [10], Floodlight [11]), have largely assumed the role of the NOS as the facilitator between benign OpenFlow applications and the data plane. The problems that may arise when an OpenFlow application contains flaws, vulnerabilities, or malicious logic that may interfere with control layer operations have remained largely unaddressed by existing OpenFlow controller implementations. Indeed, we will show examples of various OpenFlow applications that, when implemented with the basic flaws commonly found in prototype applications, lead to the crash of the control plane and effective loss of the network itself. These examples serve to both establish the importance of solid network application development and illustrate an important need for robust control layer safeguards to ensure network applications remain within bounds that protect the OpenFlow stack.

We introduce ROSEMARY as a NOS that integrates key safeguards that extend the control layer to not just facilitate OpenFlow applications, but also to impose an independent sandbox around

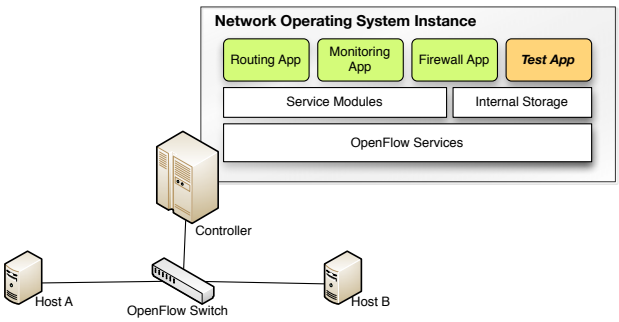

Figure 1: NOS evaluation environment

each network application. ROSEMARY's objective is to prevent applications from performing operations that will otherwise corrupt other widely used OpenFlow controllers. We refer to ROSEMARY's containment strategy as a *micro-NOS* architecture, in which each OpenFlow application is spawned within an independent instance of ROSEMARY, wherein it is monitored and constrained with resource utilization controls that prevent it from interfering with the broader operations of the OpenFlow stack.

A micro-NOS implements several key functional design requirements that combine to provide application containment. First, it provides process context separation from the control layer by spawning a network application in a separate process context, with the NOS interface libraries and data abstractions necessary for the application to operate. Second, the micro-NOS incorporates resource monitoring services that track and recognize consumption patterns that may violate resource utilization policies imposed upon the application. Third, each micro-NOS instance operates within a permission structure of the ROSEMARY NOS, that limits which libraries are incorporated into the micro-NOS, thereby constraining the ability of the network application to access interfaces or resources of the NOS kernel. Finally, ROSEMARY's design pays careful attention to the performance considerations that are imposed with its micro-NOS compartmentalization.

We present two optimization mechanisms that can be utilized in ROSEMARY deployments to enable a balance between the need for strict robustness controls and the need for network application performance. *Request pipelining* represents a series of task, IPC, and task-to-CPU core-pinning optimizations that enable substantial performance optimization in ROSEMARY's packet processing pipeline. We further introduce a *trusted execution* mode for network applications, which can be employed with well-vetted applications to remove the micro-NOS IPC overhead. With or without trusted execution services, ROSEMARY's performance is substantially superior to many leading OpenFlow controllers.

To summarize, the paper's contributions include the following:

1. We survey a range of critical robustness issues found in several existing OpenFlow controllers. We identify the underlying vulnerabilities in the controller operations that enable the network application layer to destabilize the control plane.

2. We present the design of ROSEMARY, which introduces the use of a micro-NOS sandboxing strategy to safeguard the control layer from errant operation performed by the network application. ROSEMARY's resilience in the presence of malicious or faulty network applications that would otherwise crash other OpenFlow controllers is derived from three key services: context separation, resource utilization monitoring, and the micro-NOS permissions structure that limits the network application's access to library functionality.

3. We design and implement a prototype of ROSEMARY, which we argue offers a robust application execution environment, while also enabling execution modes that can run vetted network applications with very high performance metrics relative to existing OpenFlow controllers. We have evaluated the prototype system under a range of diverse test cases to measure its performance, finding that ROSEMARY can sustain more than *10 million* flow requests per second.

Roadmap. The remainder of this paper is organized as follows. In Section 2, we provide a few motivating examples and discuss some open research problems. In Sections 3 and 4, we present the design and implementation of our system, respectively. Next, we provide evaluation results demonstrating the robustness and scalability of our system in Section 5. We also discuss system limitations (Section 6) and related work (Section 7). Finally, we conclude with a brief summary in Section 8.

2. MOTIVATING EXAMPLES AND RESEARCH QUESTIONS

2.1 Few Motivating Examples

We have witnessed the emergence of several open-source and commercial efforts to develop NOSs including POX [30], Floodlight [11], OpenDaylight [23], and NOX [14]. These controller designs have formed the basis of many commercial SDN network deployments and academic research efforts. Additional results with Beacon and POX are provided in the appendix. Until recently, most of these designs (from industry and academia) have been primarily focused on efficiently handling data plane requests. Security and robustness of OpenFlow controllers has arguably been an understudied area with some recent notable exceptions [29, 19, 36]. However, these unfledged research efforts are still very much in flux and require validation through large scale deployments. Furthermore, they have largely ignored controller robustness, i.e., the threat vector of malicious OpenFlow applications directly attacking the controller.

In this section, we motivate the need for both network controller robustness and ROSEMARY through tangible examples that underscore the vulnerability of contemporary OpenFlow controllers to malicious OpenFlow applications.

Test Environment. To test the robustness and the security of a NOS, we have set up a simple test environment, as shown in Figure 1. We choose Floodlight and OpenDaylight as our main target OpenFlow controllers, because they are two of the most popular and emerging network operating systems. Two hosts (Host A and Host B) are connected to each other through an OpenFlow switch (i.e., data plane), and this switch is controlled by the OpenFlow controller. Here, we run four network applications on the OpenFlow controller: (i) a simple routing application, (ii) a network monitoring application, (iii) a firewall application, and (iv) a test application. We modify this test application to evaluate the robustness and the security of the NOS.

2.1.1 Floodlight Case Study

We describe results from three different evaluations, including two misbehaving applications for testing robustness and another rogue application that affects security The objective of the first application is to silently crash a Floodlight instance. This application

simply calls an exit function, when it is invoked by Floodlight. The next application, whose code is illustrated below, creates multiple memory objects. We will test how this application consumes memory resources allocated to a Floodlight instance. The final application, illustrated through source code on the right, modifies the internal data structure of a Floodlight instance. In this case, we assume that this application is installed by an attacker, and that it deletes some network links in an attempt to confuse other applications.

Testing Floodlight Robustness. In this case, we let the test application conduct two unintended operations: (*i*) *exit a program suddenly* and (*ii*) *continuously allocate memory space*. These operations, when performed in the wrong context, result in a crash of the Floodlight controller.

In the first of these examples, the developer inadvertently calls a system exit or return (with exit) function. Figure 2 (in next page) shows the result: when a buggy application accidentally calls an exit function, we observe that the Floodlight instance is also killed.

For the second case, a memory leakage is presented in Figure 3 (in next page). Here, the developer creates a linked list without bounds checking and allows the list to grow without limit. The Floodlight instance does not limit memory allocations by its applications, finally resulting in the controller (technically, the JVM) crashing with an out of memory error.

As both of these errors result from programming errors, we consider these to be misbehaving, and not necessarily malicious, applications.

Testing Floodlight Security. We modify the test application to access an internal data structure in Floodlight, which contains network link information, and change certain values representing network links. To show the effect of this test, we let the monitoring application (shown in Figure 1) periodically access a Floodlight data structure (i.e., network link table) and display network link information. When our test application modifies values in the data structure (i.e., network link information), the monitoring application presents the wrong information. To perform this test, we simply create a network environment with Mininet [20], which consists of two switches connected to each other. The network link information before and after the attack is presented in Figure 4 (top and bottom) to illustrate how a simple rogue application can easily confuse other network applications.

2.1.2 *OpenDaylight Case Study*

Next, we evaluate the recently announced OpenDaylight controller [23], and check if it the problems that are similar to the Floodlight controller. Due to space we simply present the case of crashing a NOS instance by calling a system exit function, and the test result is shown in Figure 5. We find that OpenDaylight has similar robustness issues, despite its recent release (i.e., early 2014).

2.1.3 *POX Case Study*

Next, we conduct an experiment that results in the POX controller [30] being killed. In this test, we run a monitoring application and a test applications which we crash simultaneously using Mininet [20]). Here, our test application (shown in Figure 6) simply calls a system exit function (i.e., sys.exit(0) function), when it is launched, and the monitoring application periodically collects network statistics information. The result shown in Figure 7, illustrates how a crash of the test application causes both the monitoring application and the controller to terminate.

2.1.4 *Beacon Case Study*

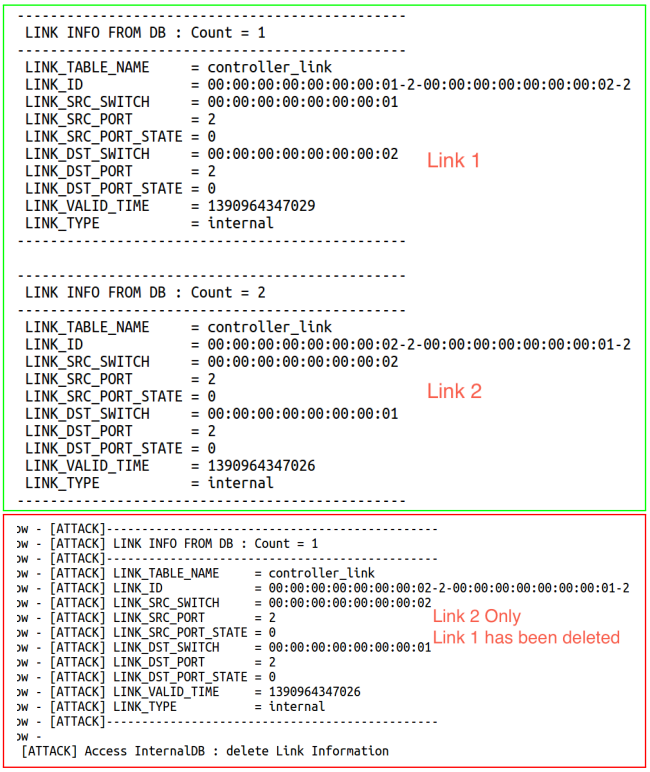

Figure 4: Network link information from Floodlight before (top) and after (bottom) attack. Only 1 link remains after the attack.

Finally, we conduct the same robustness tests (i.e., crash and memory leak) with Beacon [10], and the results are presented in Figure 8 and Figure 9. This second set of studies illustrate that these problems are not only problems of a specific NOS (i.e., Floodlight), but also prevalent in other NOSs. These examples illustrate how simple applications (in under 10 lines of code) can crash an OpenFlow controller rather easily.

We do not include exemplar security attacks on OpenDaylight, Beacon, and POX due to space limitations, but similar vulnerabilities exist. While we assume and hope that trusted network experts design and implement network applications, our objective here is to motivate the need to incorporate safeguards into the control layer

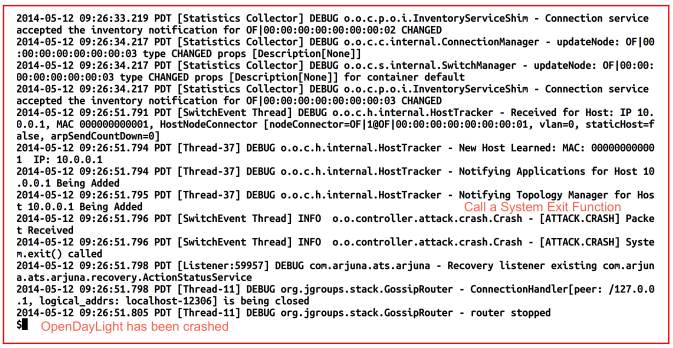

Figure 5: OpenDaylight crash result

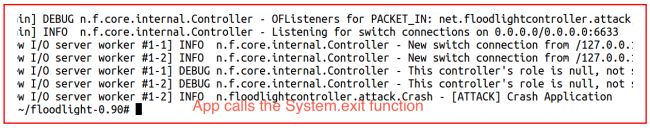

Figure 2: Floodlight crash result

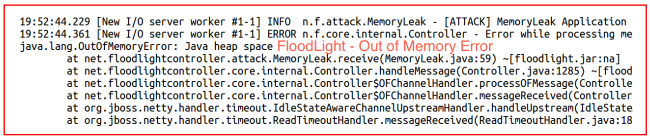

Figure 3: Floodlight memory leakage result

```
def handle_PacketIn(event):
   packet = event.parsed
   inport = event.port
   .....
def launch():
  core.openflow.addListenerByName("PacketIn", handle_PacketIn)
  print '[ATTACK] Crash Application'
  sys.exit(0)
```

Figure 6: POX crash source code

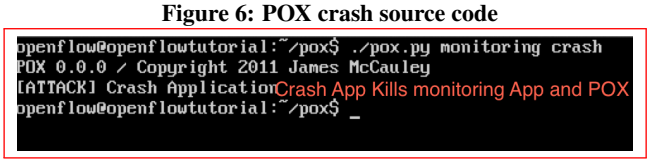

Figure 7: POX crash result

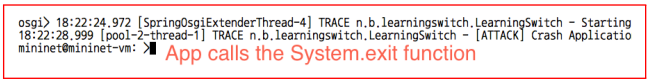

Figure 8: Beacon crash result

Figure 9: Beacon memory leakage result

design to provide a degree of resilience to such deviant network application behavior.

2.2 Why Are NOSs NOT Robust?

R1. No Separation of Applications from Network OS. Perhaps the most critical problem affecting existing NOSs is that they run applications in the same privilege zone where a NOS resides. For example, in the case of NOX, it runs a network application as a function invoked from the main module of NOX [14]. We assume that the reasoning behind this design choice is to improve performance by reducing potential overhead due to increased interprocess communication (IPC) and additional I/O between the main module and the application.

However, this design choice poses serious problems when coupled with misbehaving applications. If a network application crashes, it will also kill a NOS. Of course, one could pass the burden to skilled programmers and static program analysis methods to ensure correctness. In practice, it is very hard to guarantee that an application is flawless across all inputs and execution scenarios.

Runtime enforcement of application robustness and security constraints, simplifies SDN application development and fosters greater SDN adoption through third-party SDN application stores [16] that are modeled after popular mobile application stores. This motivates the need for rethinking NOS design with a critical eye toward protection from buggy and untrusted network applications.

R2. No Application Resource Control. SDN applications can interfere with NOS and other co-resident applications in other ways besides crashing the controller. For example, they may consume more than their fair share of system resources (memory or CPU) such that other applications and the network controller cannot conduct necessary operations.

This problem stems from the lack of resource controls for each application. Traditional operating systems have mechanisms to limit the resource usage and provide fairness across applications (e.g., schedulers, ulimit, etc.) However, most existing NOSs do not provide such capabilities and allow a network application to obtain resources without constraints, thereby undermining the robustness of the NOS.

R3. Monolithic NOSs–A Case for micro-NOS. If we consider the main module of a NOS as the kernel of operating system, we can then view the design of most NOSs as monolithic architectures that contain the most important services running as an agglomerated kernel with all privileges. Most NOSs provide diverse services, such as internal storage and event handling [17], to help developers easily implement network applications. Usually, these services are integrated as functions into the main module of a NOS.

This sort of monolithic architecture can support high-performance at the cost of added complexity. However, as NOSs evolve, they become burdened with more and more functionality for legacy application support and to ease development of new network applications. For example, the latest version of Floodlight provides much more functionality than first generation of the NOX controller. We make the case that the design of NOSs need to be fundamentally reconsidered with first principles from the operating systems community. Specifically, we argue for efficient containment of applications, improved monitoring of resource utilization for policy enforcement and an effective permission structure.

2.3 Why Are NOSs NOT Secure?

R1. No Authentication. Most contemporary NOSs assume that network applications are trustworthy and do not implement any mechanisms to validate the authenticity of their authorship. For example, in the case of commodity operating systems applications have the ability to be signed, and digital certificate validation is used to notify users when they attempt to run applications that fail validation. Similarly, ROSEMARY incorporates a process for authenticating application developers. Here, we rely on recent research efforts on a new secure NOS that supports digital application authentication [27].

R2. No Access Control. Traditional operating systems (OSs), such as Linux and Windows, do not allow an application to directly access resources managed by the OS. When an application needs access to an OS resource, it requests permission from the operating system (i.e., kernel layer). However, in the case of a NOS, as shown in the example above, a network application can easily access a resource managed by a NOS. This problem stems from the lack of access control methods for system resources.

2.4 What About Performance?

Since a NOS needs to handle many network requests from multiple data planes, most NOSs are designed to treat as many network

requests as possible [11, 14]. In our design, to implement a more robust and secure NOS, we must ensure that the added costs of these extensions do not introduce overheads that are prohibitive.

3. SYSTEM DESIGN

3.1 Design Considerations and Philosophy

Considering the issues described in Section 2, we summarize the guiding principles that inform the design of ROSEMARY as follows.

- network applications must be separated from the trusted computing base of the NOS
- resources for each network application needs to be monitored and controlled
- shared modules of NOSs need to be compartmentalized
- network applications must explicitly possess capabilities to access resource of a NOS
- the system must be adjustable to balance the need for robustness safeguards and the need for higher performance.

In this section, we present the design of a new NOS, ROSEMARY, that follows from these principles. The overall architecture of ROSEMARY is illustrated in Figure 10. Technically, ROSEMARY is an application program running on a generic operating system (e.g., Linux), much like other NOSs.

ROSEMARY employs a specialized application containment architecture that is inspired by prior work on microkernels [1], and exokernels [9]. These architectures sought to enable increased security and stability by reducing the amount of code running in kernel mode[1]. Exokernels seek to *separate protection from management* and they do so by using "secure bindings" that reduce the shared portions of OS kernels and implementing "resource revocation" protocols. This approach minimizes the running overhead of an application and reduces hardware abstraction by moving abstractions into untrusted user-space libraries, allowing application visibility into low-level resource information.

We believe that a robust and secure SDN controller can benefit from several of these ideas. ❶ First, a NOS needs to provide a clear view of the data planes to applications: the abstraction of a NOS should be minimal. ❷ Second, applications and libraries can be buggy. Hence, important modules for a NOS should be compartmentalized to maximize the reliability of a network operating system. ❸ Third, each application has its own purpose, and thus each application requires specialized functions and libraries. ❹ Finally, a NOS needs to be minimal and lightweight to guarantee that it can handle large numbers of network requests with low latency.

3.2 System Architecture

As illustrated in Figure 10, ROSEMARY consists of four main components: (i) a data abstraction layer (DAL), (ii) the ROSEMARY kernel, (iii) system libraries, and (iv) a resource monitor. DAL encapsulates underlying hardware devices (i.e., network devices) and forwards their requests to the upper layer (i.e., ROSEMARY kernel). A key objective of DAL is to marshal requests from diverse network devices. For example, while OpenFlow is the dominant SDN interface between the data plane and the control plane, it is conceivable that future controllers would want to support a diverse set of SDN protocols (e.g., XMPP [28], DevoFlow [7]) in an application-agnostic manner.

The ROSEMARY kernel provides basic necessary services for network applications, such as resource control, security management, and system logging. They are basic services to operate network applications, and we design this component as thinly as possible to expose a clear view of the data plane to an application. ROSEMARY provides diverse libraries for applications, and each application can choose necessary libraries for its operations. When it selects libraries, each application is required to solicit permission from ROSEMARY kernel. The resource monitor (RM) is used to track the respective resource utilization of running applications and terminate misbehaving applications.

The components described here will be used in addressing the aforementioned design issues. Below, we discuss in detail how each issue is addressed with specific component(s).

1) Separating Applications from a Network OS Kernel. As we presented in Section 2, one of the primary reasons behind the fragility of NOSs is their tight coupling with applications. To address this issue, we need to have a clear separation between applications and the NOS. A simple means to ensure such isolation is to spawn applications as independent processes. In ROSEMARY, each new network application will be invoked as a new process, which connects to the ROSEMARY kernel process instance through a generic IPC method.

Imagine a scenario where an application periodically requests several services from ROSEMARY (e.g., network topology information, switch statistics). If we provide each of these services only through an IPC channel, the overhead caused by such communication could be considerable. To mitigate such overhead, we let an application use libraries provided by ROSEMARY to directly communicate over the network. This design is inspired by the Exokernel [9]. For example, if an application wants to access a data structure, it can attach itself to the storage library as shown in Figure 10. Of course, each application is required to obtain the specific capabilities prior to using the library.

2) Compartmentalizing Network OS Kernel Modules. The notion of privilege separation needs to be considered not only between the application and a NOS instance, but also in the design of the NOS itself. To the best of our knowledge, leading NOSs (e.g., Floodlight, NOX, and POX) have all implemented necessary functions in a single protection zone. We refer to such a single-protection-zone NOS as a *monolithic kernel architecture*.

While a case could be made that the monolithic architecture improves performance by minimizing communications across protection zones (IPC), the inherent complexity of such architectures can make reasoning about its security and isolation challenging. The first generation of NOSs (e.g., NOX classic version [14]) only supported elemental functionality – receive flow requests from the data plane, deliver them to applications, and enforce flow rules to the data planes – to make their design lightweight and simple. However, this is not so true anymore. Current NOSs are bundled with many component libraries to simplify application development. For example, Floodlight provides network status information, and an application developer can simply query this information from Floodlight instead of reimplementing this logic.

Adding complexity to any system inherently increases its vulnerability to failures and misbehavior. A NOS that conducts the

[1]A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality. – Liedtke's minimality principle

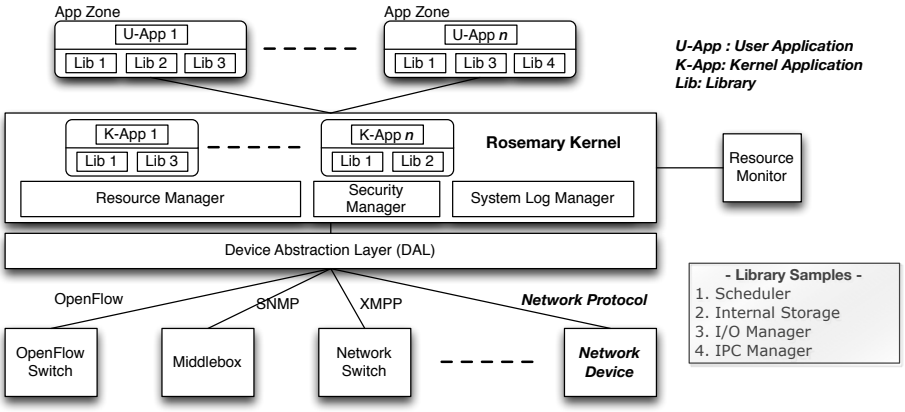

Figure 10: Overall architecture of ROSEMARY comprising of four key components: (i) a data abstraction layer (DAL), (ii) the ROSEMARY kernel, (iii) system libraries, and (iv) a resource monitor. ROSEMARY provides diverse libraries for applications, and each application can choose necessary applications for its operations. Applications may be run in user-mode for robustness or kernel-mode for maximal performance.

mission-critical operation of managing networks should be designed to be unsusceptible to such calamities.

To improve controller resilience, we separate services in the ROSEMARY kernel into different pieces (i.e., processes), making it analogous to a *microkernel architecture*. We call this design a micro-NOS. A NOS with this architecture has several advantages over a NOS using the *monolithic architecture*. First, it makes a network operating system lightweight by enabling on-demand invocation of services. For example, if a NOS only needs to support an application for simple network switching, it simply needs to distribute data plane events to the application; thus it is not necessary to run other network services.

This design increases the robustness of a NOS, because it only runs necessary services. Second, services in the ROSEMARY kernel communicate each other through an IPC channel, and the implication is that if a service crashes, other services are immune to this crash.

3) Controlling Application Resource Utilization. Although network applications are detached from the core of the ROSEMARY network operating system and cannot crash it, it is possible that a buggy or a malicious network application can interfere with the operation of co-resident applications. As shown in Section 2, an application that keeps allocating memory can consume all of the available memory in a host, thereby affecting other applications.

To mitigate this effect, we need to differentiate each application's working space and limit resources that each network application can use, much like a generic operating system does. ROSEMARY also provides similar functionality to control the resource usage of each network application through its *resource manager* (RM) component. The vantage point of the RM provides unique ability to control diverse resources used by SDN applications, but here we focus on critical system resources that are tightly coupled with the robustness of a network operating system and co-resident applications. The resources that ROSEMARY currently considers include the following: (i) CPU, (ii) memory, and (iii) file descriptor.

The algorithm to control the resources is presented in Figure 11 (right). The maximum resources assigned to each application are managed by a table. This table, illustrated through the resource table in Figure 11, maintains two values for each resource item: (i) a *hard limit* and (ii) a *soft limit*. The hard limit specifies that an application cannot obtain more resource than this value. For example, if a hard limit value for the memory item is 2 GB, then the application is terminated if it attempts to allocate more than 2 GB of memory. The soft limit defines a value that each application may pass but is not recommended. Violations of the soft limit result in an alert that is passed back to the application and reported to the network operator.

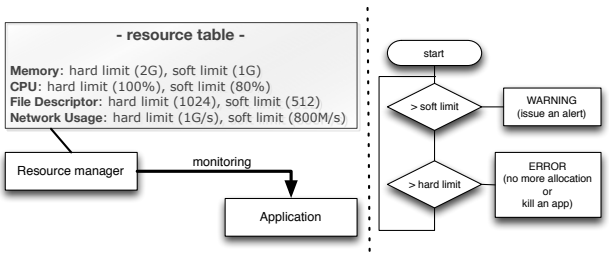

Figure 11: Application resource monitoring and control

4) Providing Access Control and Authentication. The aforementioned design choices are primarily intended to improve system robustness; however, we still need to address the security concerns described in Section 2. For example, a malicious application may access internal data structures and modify them without restriction, thereby confusing other network applications.

We address this issue by employing a sandbox approach, which we have named *App Zone*. Its detailed architecture is presented in Figure 12, and it essentially runs an application within a confined environment to monitor and control the application's operations. When an application makes privileged system calls (e.g., spawns processes or accesses internal storage modules), such operations are interposed by our management module which we call the *system call access check module*. This enables the operator to control the set of operations that may be performed by applications.

In addition, App Zone examines whether this application is authorized or not by investigating its signed key (*application authorization module*). ROSEMARY provides a public key, and all application developers are required to sign their applications with this key. Such an approach has also been adopted by other related work

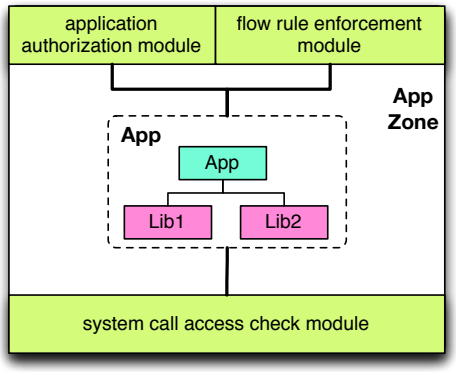

Figure 12: Internal architecture of Application Zone

[26]. Besides these issues, App Zone monitors flow rule enforcement operations initiated by an application to investigate whether there is any abnormal behavior. For example, if an application tries to enforce too many flow rules that cannot be inserted into the data plane, its operation is considered abnormal. In this case, the *flow rule enforcement module* monitors this operation, and reports if it finds these abnormal behaviors. In the current state, we have a threshold-based detector, which generates an alert when it finds an application that tries to deliver more flow rules than a predefined threshold value. This threshold value will be determined based on the capability of the data plane at boot time. Specifically, this module checks the maximum number of flow rule entries for each data plane, and it sets the corresponding threshold value to be a certain percentage of this number (currently, we use 90%).

5) Monitoring the Network OS. When ROSEMARY is invoked, its core services are monitored by an external monitoring process (*Resource Monitor* in Figure 10). If this monitor process detects any anomalous behaviors in ROSEMARY (e.g., service crash or memory leakage), it will react to them are summarized in Table 1.

Behavior	*Reaction*
Service crash	restart the crashed service
Multiple services crash	restart ROSEMARY
Memory leakage of a ROSEMARY service is detected	restart a service causing memory leakage
Kernel App crash	restart ROSEMARY services (if necessary), and restart the app as a user app

Table 1: Reaction strategies against anomalous behaviors of ROSEMARY

When the monitor process detects any anomalous behaviors, it also records them into a logging system. This record is useful for further analysis of anomalous behaviors as well as for improving and hardening ROSEMARY.

6) Safely Restarting the Network OS. When the Resource Monitor process detects serious problems (e.g., crash of critical services) it immediately restarts ROSEMARY services. At this time, ROSEMARY decides whether or not it must restarts all services and network applications. Since sometimes it is not easy to discover why ROSEMARY crashed (i.e., which service or application caused ROSEMARY to crash), we need to carefully restart each service. Inspired by the Safe Mode booting of the Windows operating system that boots up only the basic and necessary services of Windows, we have designed a *safe mode booting process* for ROSEMARY that is similar in spirit.

When the Resource Monitor decides to boot up ROSEMARY in Safe Mode, it only runs the basic services and network applications that are defined by the administrator. At first, it invokes basic ROSEMARY kernel services: resource manager, system log manager, and security manager. Then, it runs network applications in the order specified by the administrator. Since it is very possible that a single misbehaving application is the main cause of the crash, we need to run only necessary network applications at the boot stage. Once this initial boot is complete, if there are any kernel-applications running, it changes their modes to be user-level applications in order to guarantee robustness. After booting up necessary services and applications, the Resource Monitor analyzes log files to investigate the reason behind the problem. If the cause of the crash can be attributed to a single application, it reports this log indicator to the administrator.

7) Performance Considerations. Separation of applications from a NOS instance is good for robustness and security, but it imposes additional latency overhead due to added IPC. To minimize the overhead and guarantee required performance, we employ two mechanisms: (*i*) *request pipelining* and (*ii*) *trusted execution*.

Request Pipelining - Pipelining is a well-known technique that splits a task into multiple subtasks and executes them with as many execution units as subtasks. Though it adds latency overhead to pass data between subtasks, it can greatly improve throughput. ROSEMARY overlaps the subtask split for pipelining with the kernel/application separation for robustness, named request pipelining, as shown in Figure 13. As such, from the perspective of pipelining, the kernel and applications are pipelined subtasks and their communication is data transfer between subtasks. When ROSEMARY receives packets from switches and routers, the kernel thread processes the controller protocol (e.g., OpenFlow), prioritizes the packets, passes them to applications through an IPC (inter-process communication) primitive, and completes the first stage of request pipelining. The IPC primitive, a domain socket in the case of the current implementation, works both as a pipelining latch for throughput improvement and as a boundary of separated execution between the kernel and application for robustness. Receiving the packets through the IPC primitive, the application that will process the packets starts the second stage of request pipelining. In Section 5, we show how request pipelining yields high packet processing throughput while also providing robustness in ROSEMARY.

For further performance improvement, ROSEMARY carefully pins down the kernel process and application processes onto CPU cores so that communication between pipelined stages is assigned to adjacent CPU cores for lower CPU interconnect overhead. In addition, interrupt signals from network cards are configured to be routed to the CPU cores that are running the kernel process in order to eliminate possible interrupt rerouting among cores. Lock-free data structures, such as a lock-free FIFO for queuing packets, are heavily used to avoid locking overhead as well.

Trusted Execution - While request pipelining provides high throughput and robustness, ROSEMARY offers another application execution mode for latency-critical applications, named trusted execution. At the discretion of system operators, an application can be integrated with the kernel and run in the kernel process; we call this a *Kernel-application*). The application should be seriously sanitized, because this execution mode compromises robustness for latency

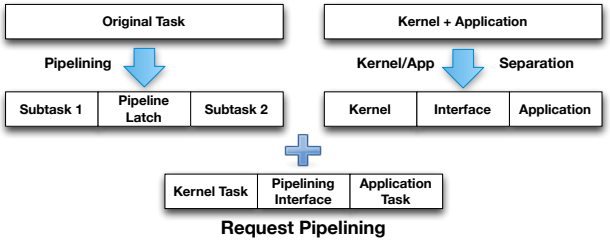

Figure 13: Overall approach of request pipelining. IPC is used as a pipelining latch for throughput improvement and as a boundary of separated execution between the kernel and application.

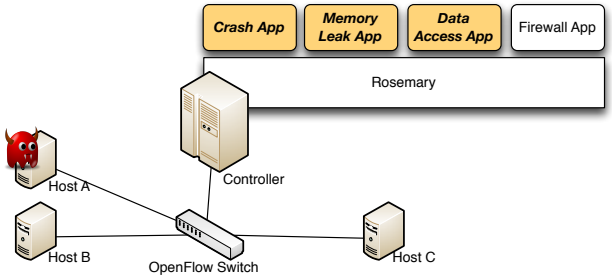

Figure 14: Test environment for evaluating the robustness and security of ROSEMARY

reduction. If the application crashes and kills a kernel service, the Resource Monitor restarts a kernel service promptly but does not allow the application to restart in trusted execution mode. Instead, the application runs in a normal mode (i.e., User-Application) with request pipelining.

4. IMPLEMENTATION

We have implemented a prototype of ROSEMARY to evaluate its robustness, security, and performance. We implement the prototype as a C application consisting of approximately 20,000 lines of code. ROSEMARY libraries are also written in C. The system currently supports both the OpenFlow 1.0 specification [12] and the OpenFlow 1.3 specification [13] in the data abstraction layer (DAL), and we are planning to support the XMPP protocol [37] in the near future.

We employ standard *pthread* libraries (i.e., POSIX pthread [4]) to implement the basic kernel services and applications to increase the portability. In addition, each network application can run multiple working threads at the same time to increase its performance, and the thread package (i.e., pthread) has also been used in this case.

In its current stage, APIs for developing an application (both user application and kernel application) are written in C, and thus network applications are also written in C. We are currently developing APIs in additional languages, such as Python and Java, which will provide developers with additional options. We have also implemented a tool to generate both types of applications automatically. Note that a network application developer need only implement an application once, and it can be instantiated as either a user or kernel application.

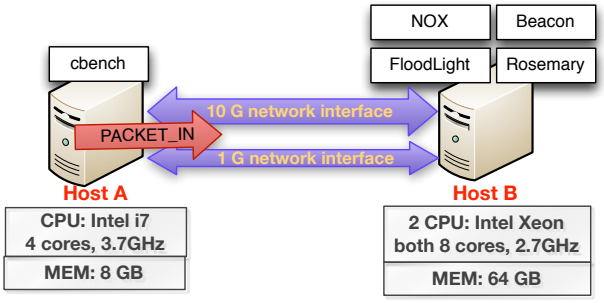

Figure 15: Test environment for measuring throughput of each NOS

5. EVALUATION

5.1 Evaluation Environment

Robustness and Security Testing Environment. Figure 14 presents the evaluation environment used to test the robustness and the security of ROSEMARY. In this environment, we run three applications that we have tested in Section 2. The first application - Crash App - will simply call an exit function to kill itself. The second application - Memory Leak App - will allocate memory objects continuously. The third application - Data Access App - will try to access an internal data structure, which is not allowed. In addition, we run a normal network security application - Firewall App, which simply blocks malicious packets from Host A - to investigate whether this application is affected by other applications.

Performance Testing Environment. To measure and compare the throughput of each NOS, we use the test environment shown in Figure 15. Here, we have two independent hosts: (i) a client generating data plane requests and (ii) a server operating each NOS. The hardware specifications of each host are illustrated in Figure 15. Note that each host is connected with two network interfaces: (i) 1 Gbps and (ii) 10 Gbps NICs, respectively. We turn on each network interface one at a time to test four different NOSs - Floodlight [11], NOX [14], Beacon [10], and ROSEMARY). For each software, we have installed their most recent version[2].

In addition, to minimize any unintended overhead, we run a single NOS at a time, and turn off any functions that may cause additional overhead (e.g., system logging) for each application. A simple *learning switch* application is used for this test, because all NOSs provide a similar application by default. For generating data plane requests, we use the cbench tool [25], because it is widely used in measuring the throughput of NOSs.

5.2 Robustness Test

We first present the test result for ROSEMARY, when it runs an application that simply calls an exit function (i.e., Crash App in Figure 14)). As shown in Section 2, Floodlight, POX and Beacon are terminated along with similar test applications. However, since core parts of ROSEMARY are clearly separated from applications, they are not affected by this crash and other applications continue to run. This result is presented in Figure 16, which shows that a firewall app can block attack trials from Host A whose IP is 10.0.0.1 (Figure 16 (red rectangle in left bottom)), even if the Crash App has been terminated (Figure 16 (right bottom)). In addition, we can see

[2]Specifically, in the case of NOX, we have used the newly developed multi-threaded NOX and NOT the classic version.

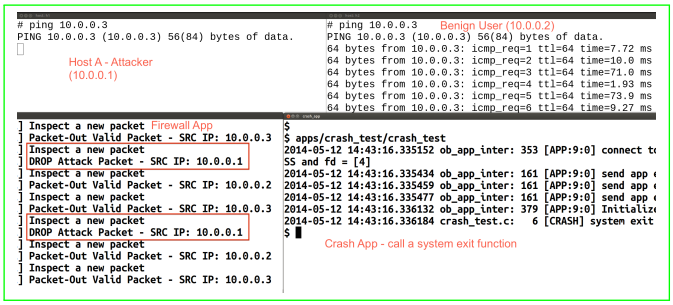

Figure 16: Application crash test for ROSEMARY. A firewall app will block attack trials (left) although the Crash App terminates with a System.exit() (right).

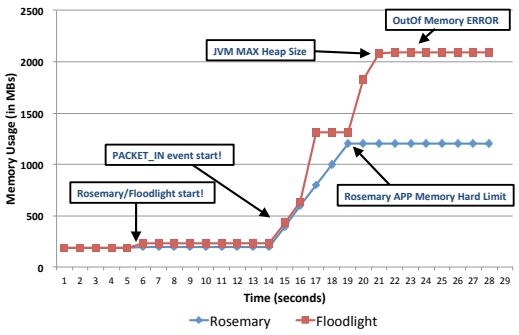

Figure 17: Memory leakage test for Floodlight and ROSEMARY. Floodlight crashes with an out-of-memory error while ROSEMARY limits memory allocation to 1.2 GB.

that Host B (a benign user whose IP address is 10.0.0.2) is able to contact Host C without any problem (Figure 16 top-right).

In addition, we run an application that keeps allocating memory objects (i.e., our Memory-Leak App) to verify how ROSEMARY handles this condition. We also run the same application on Floodlight to clearly compare the reaction of each NOS, and the result is shown in Figure 17. When an application starts its operation (at time 14), we can easily observe that the memory footprint of each NOS is increasing. Here, since we set a hard limit of the memory usage for the ROSEMARY application at 1.2 GB, the test application running on ROSEMARY cannot allocate more than 1.2 GB of memory (at time 19). However, the memory footprint of the application running on Floodlight keeps continuously increasing until it reaches its maximum allocation size (at time 21) at which point Floodlight issues an out-of-memory error alert.

5.3 Security Test

We repeat the security experiment from Section 2 to test the security of ROSEMARY. We created an application (i.e., Data-Access App) that accesses and modifies an internal data structure. Unlike the case of Floodlight, ROSEMARY checks whether an application has a right to access or modify a data structure; thus this application fails in accessing the data structure, as it cannot obtain the necessary capability. Figure18 shows a screen shot in which the Data-Access App tries to modify a data structure inside ROSEMARY, which is not allowed. We can clearly see that the access attempt has been rejected by ROSEMARY.

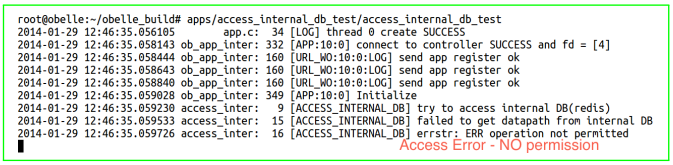

Figure 18: Data-Access App tries to modify a privileged data structure inside ROSEMARY, which is not allowed. This access attempt is rejected by ROSEMARY.

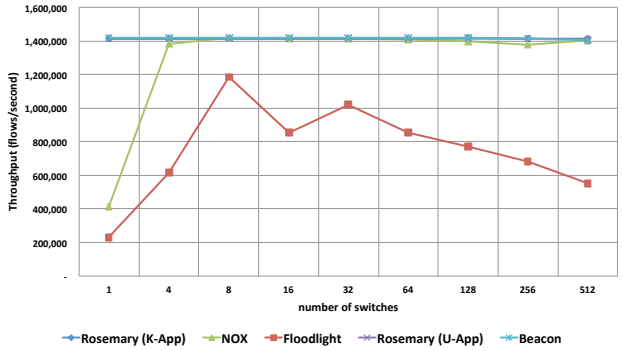

Figure 19: Throughput comparison between ROSEMARY and other NOSs (1G interface, varying the number of switches, and the number of threads is 8)

5.4 Performance Test

Throughput. We measure the overall throughput of ROSEMARY, and compare it with other NOSs - NOX [14], Beacon [10], and Floodlight. Here, we have optimized each NOS based on its recommendations. In the case of Beacon [10], we have followed the instructions described in a recent work [10]. Floodlight also provides instructions for benchmark [24]. Although NOX does not provide such an instruction, we use all recommended libraries (e.g., boost library [3]) to improve its performance. In addition, in the case of ROSEMARY, we separately measure two cases: a test application is implemented as (i) User-Application (denoted as ROSEMARY U-App) and (ii) Kernel-Application (denoted as ROSEMARY K-App).

Figure 19 and Figure 20 show that the overall throughput of each NOS when the data plane (i.e., Host A in Figure 15) and the control plane (Host B in Figure 15) are connected through a 1G network interface. Figure 19 presents a case in which we vary the number of connected switches (i.e., data planes) when the number of working threads for each network operating system is fixed at 8. In this case we can clearly observe that ROSEMARY (both U-App and K-App) and Beacon nearly saturate the hardware limitation of network bandwidth (i.e., 1 GB). In the case of NOX, it also saturates the hardware limitation when it is connected to more than 4 switches. However, the throughput of Floodlight is relatively low compared with others.

Next, we vary the number of working threads from 1 to 16 to understand how the performance of each NOS depends on the number of working threads, and the results are presented in Figure 20. In this case, we set the number of the connected switches as 64. Similar to the result of Figure 19, ROSEMARY (both U-App and K-App) and Beacon nearly hit the hardware limitation, even if they only have a small number of working threads. NOX also saturates the hardware limitation, but it requires more than 5 working threads. Floodlight shows much worse performance than other NOSs, and

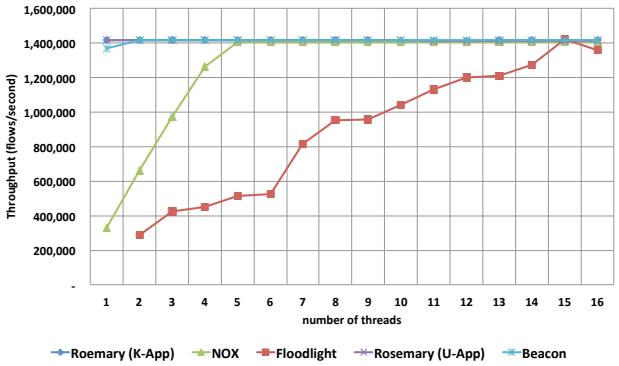

Figure 20: Throughput comparison between ROSEMARY and other NOSs (1G interface, varying the number of threads, and the number of switches is 64)

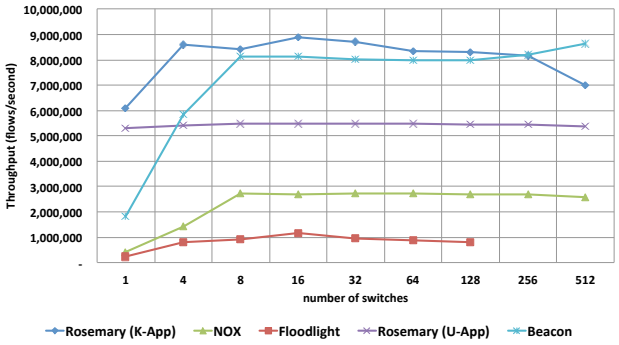

Figure 21: Throughput comparison between ROSEMARY and other NOSs (10G interface, varying the number of switches, and the number of threads is 8)

it reaches the saturation value when it has more than 15 working threads.

We changed the network interface between the data plane and the control plane to a 10G interface, and we repeated the same tests in order to discover the maximum throughput of each NOS. Figure 21 presents the results of throughput when we vary the number of switches while fixing the number of working threads at 8. In this case, we can clearly find that ROSEMARY outperforms most of the the other NOSs. For example, ROSEMARY (K-App) shows 8 to 25 times better performance than Floodlight, and 3 to 14 times better performance than NOX. Although we only consider ROSEMARY (U-App), it also shows 5 to 22 times better performance than Floodlight, and from 2 to 12 times better than NOX.

When we vary the number of working threads with the 10G interface, the test results (in Figure 22) still show that ROSEMARY outperforms most of the other network operating systems. At this time, ROSEMARY (K-App) can handle more than 10 million network requests from the data planes. We believe these results show that ROSEMARY can support a large scale network environment. For example, recent work in [2] surveys how many new flows will hit a centralized controller, and its estimation result shows that around 10 million new flows can be issued by a data center in the worst case. In this case, ROSEMARY can handle this large number of requests and provides capabilities for robustness and the security at the same time. Beacon outperforms ROSEMARY in some cases - when there more than 11 working threads– we suspect this is due

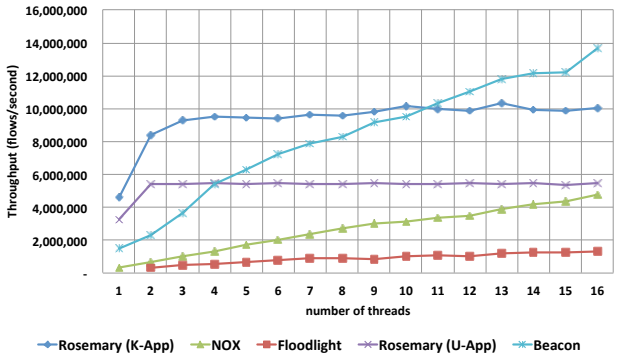

Figure 22: Throughput comparison between ROSEMARY and other NOSs (10G interface, varying the number of threads, and the number of switches is 64)

to of ROSEMARY's CPU pinning overhead. However, in our test, we find that CPU pinning overhead is nearly ignorable, and thus we think that the performance difference between ROSEMARY and Beacon is caused by the scheduling method of each application. We plan to investigate this issue more deeply in future work.

6. LIMITATIONS AND DISCUSSION

Although our work considers diverse issues concerning the security and robustness of NOSs, there remain lingering challenges that are attractive avenues for future research.

First, ROSEMARY primarily focuses on potential threats from a buggy or malicious application, but does not address threats from vulnerabilities in the host operating system. For example, if a malware infects the host running a NOS, it can abuse the NOS quite easily. We believe that such threats would be addressed through the use of auxiliary host and network intrusion monitoring systems, as well as by following best practices in host and network management, e.g., regular patch management and isolating the management interface of the network controller to selected trusted internal hosts.

Second, if a remote attacker mounts a *control flow saturation attack* against ROSEMARY [33], the impact of flow request floods may degrade the SDN controller and application performance. Here, the adversary attacks the SDN stack by crafting packet streams to saturate the data plane to control plane communications. This kind of attack is beyond the scope of our current work, but is addressed by systems such as [34]. These SDN extensions are complementary and could be integrated into ROSEMARY in future work.

Third, ROSEMARY kernel-mode applications provide improved performance at the cost of robustness and security. If an operator strictly requires robustness and security from the NOS, he or she may run all SDN applications as user applications. Although the user application gives up some performance (around 20 - 30% in 10G, but interestingly close to 0% in 1G) for security, we believe such trade-offs may well be worth the cost of resilience. We will investigate opportunities to further narrow this gap in future work, such as reducing the application scheduling overhead.

7. RELATED WORK

Our work builds on a rich body of prior work on designing network operating systems [14, 11, 30, 5, 15, 18, 35, 22, 10, 8]. These systems have largely focused on improving the scalability of the control plane[14, 11, 30, 10] or efficiently distributing control plane

functionality across different nodes and locations [18, 5, 35, 22, 8, 15] and provide many useful insights on devising more advanced network operating systems. While the contributions of these efforts in evangelizing SDN technology cannot be understated, we believe that robustness and security issues for NOSs have been overlooked. Our work aims to fill this hole by exploring complementary methods that improve the robustness and security of NOSs while providing performance that is comparable to leading controllers.

The problem of malicious contoller applications attacking the data plane was first raised by Porras et al. [29]. They address this problem by designing a security enforcement kernel for SDN controllers whose primary objective is to perform rule conflict detection and resolution. An extended version of the system [26] provides support for digital certificate validation of applications much like ROSEMARY. While these efforts have focused on protecting the data plane from malicious applications, our central objective is to improve the resilience of the control plane to both buggy and malicious applications.

Several recent studies have investigated security issues in SDNs [19, 33, 32]. Shin et al., discuss techniques to protect the control plane against control flow saturation attacks[33]. Kreutz et al. [19] and Scott-Hayward et al. [32] survey possible SDN security issues. These efforts inform and motivate the development of ROSEMARY. In addition, there have been parallel research efforts that have explored applying generic operating system design principles toward the design of network operating systems. Matthew et al. have proposed a new network operating system that borrows several ideas from generic operating system design [21]. Specifically, they introduce a file system concept to the network operating system which they implemented through a prototype system. However, their work does not consider robustness and security issues in NOS design. Wen et al., discuss a secure controller platform [36] that is primarily focused on implementing access control methods in NOS. ROSEMARY differs from their system in that it tackles diverse challenges (i.e., robustness, security, and performance) and illustrates how these challenges could be addressed in tandem.

Finally, another approach to addressing robustness issues is through formal methods. Canini et al.[6] use model-checking to holistically explore the state space of SDN networks and detect when they reach inconsistent network states. While model checking works well for small control programs, it is vulnerable to state explosion on large systems. Scott et al. [31], describe a new technique called *restrospective causal inference* for post-facto analysis of failures. We believe that both of these techniques are complementary to our design of a robust and secure micro-controller.

8. CONCLUSION

Balancing the seemingly irreconciliable goals of robustness, security, and performance represents a fundamental challenge in contemporary network controller design. We view this as an opportunity to advance network security research and realize our vision through the design of a new NOS called ROSEMARY.

ROSEMARY adopts a practical and timely approach to addressing this challenge through careful redesign of a NOS comprising three integral features: (i) context separation, (ii) resource utilization monitoring, and (iii) the micro-NOS permissions structure which limits the library functionality that a network application is allowed to access. In addition, ROSEMARY also provides competitive performance, while supporting its robustness and security features. Through our in-depth evaluation, we confirm that our system can effectively handle a large volume of network requests (nearly saturating hardware limitation in some cases) while ensuring the robustness of the OpenFlow stack in the presence of faulty or malicious network applications.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful and insightful comments. In this work, Seungwon Shin was supported by the ICT R&D program of MSIP/IITP, Korea (2014-044-072-003, Development of Cyber Quarantine System using SDN Techniques) project, and researchers at Atto Research Korea were supported by the ICT R&D program of MSIP/IITP, Korea (10045253, The development of SDN/OpenFlow-based Enterprise Network Controller Technology) project. Additional funding for researchers at SRI was provided by the MRC2 Project that is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. It is approved for public release, distribution unlimited.

10. REFERENCES

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for unix development. In *USENIX Conference* (1986).

[2] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (2010).

[3] BOOST.ORG. Boost c++ libraries. `http://www.boost.org/`.

[4] BUTENHOF, D. R. Addison-Wesley, 1997.

[5] CAI, Z., COX, A. L., AND NG, T. S. E. Maestro: A system for scalable openflow control. In *Rice University Technical Document* (2010).

[6] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *Usenix Symposium on Networked Systems Design and Implementation* (April 2012).

[7] CURTIS, A., MOGUL, J., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of ACM SIGCOMM* (2011).

[8] DIXIT, A., HAO, F., MUKHERJEE, S., LAKSHMAN, T., AND KOMPELLA, R. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[9] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995).

[10] ERICKSON, D. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[11] FLOODLIGHT. Open sdn controller. `http://floodlight.openflowhub.org/`.

[12] FOUNDATION, O. N. Openflow specification 1.0. `https://www.opennetworking.org/images/`

stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[13] FOUNDATION, O. N. Openflow specification 1.3. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[14] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review* (July 2008).

[15] HASSAS YEGANEH, S., AND GANJALI, Y. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (2012).

[16] HP. Hp opens sdn ecosystem. http://h17007.www1.hp.com/us/en/events/interop/index.aspx#.UuNbjGSmo6g.

[17] HUB, O. Floodlight architecture. http://www.openflowhub.org/display/floodlightcontroller/Architecture.

[18] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *The Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[19] KREUTZ, D., RAMOS, F. M., AND VERISSIMO, P. Towards secure and dependable software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[20] MININET. An instant virtual network on your laptop (or other pc). http://mininet.org/.

[21] MONACO, M., MICHEL, O., AND KELLER, E. Applying operating system principles to sdn controller design. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013).

[22] ONLAB.US. Network os. http://onlab.us/tools.html#os.

[23] OPENDAYLIGHT. Open source network controller. http://www.opendaylight.org/.

[24] OPENFLOWHUB.ORG. Benchmark configuraion. http://www.openflowhub.org/display/floodlightcontroller/Benchmarking+Configuration.

[25] OPENFLOWHUB.ORG. Cbench. http://www.openflowhub.org/display/floodlightcontroller/Cbench+(New).

[26] OPENFLOWSEC.ORG. Se-floodlight. http://www.openflowsec.org/Technologies.html.

[27] OPENFLOWSEC.ORG. Secuirng sdn environment. http://www.openflowsec.org.

[28] PLANET, E. N. Juniper builds sdn controller with xmpp. http://www.enterprisenetworkingplanet.com/datacenter/juniper-builds-sdn-controller-with-xmpp.html.

[29] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A security enforcement kernel for openflow networks. In *Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2012).

[30] POX. Python network controller. http://www.noxrepo.org/pox/about-pox/.

[31] SCOTT, C., WUNDSAM, A., WHITLOCK, S., OR, A., HUANG, E., ZARIFIS, K., AND SHENKER, S. How did we get into this mess? isolating fault-inducing inputs to sdn control software. Tech. rep., Technical Report UCB/EECS-2013-8, EECS Department, University of California, Berkeley, 2013.

[32] SCOTT-HAYWARD, S., O'CALLAGHAN, G., AND SEZER, S. Sdn security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for* (2013).

[33] SHIN, S., AND GU, G. Attacking software-defined networks: A first feasibility study. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[34] SHIN, S., YEGNESWARAN, V., PORRAS, P., AND GU, G. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (2013).

[35] TOOTOONCHIAN, A., AND GANJALI, Y. Hyperflow: A distributed control plane for openflow. In *Proceedings of the Internet Network Management Workshop/Workshop on Research on Enterprise Networking* (2010).

[36] WEN, X., CHEN, Y., HU, C., SHI, C., AND WANG, Y. Towards a secure controller platform for openflow applications. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[37] XMPP.ORG. Extensible messaging and presence protocol. http://xmpp.org.