AFRL-RI-RS-TR-2016-050

MODULAR RESEARCH-BASED COMPOSABLY TRUSTWORTHY MISSION-ORIENTED RESILIENT CLOUDS (MRC2)

SRI INTERNATIONAL

FEBRUARY 2016

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* FEBRUARY 2016	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2011 – SEP 2015

4. TITLE AND SUBTITLE MODULAR RESEARCH-BASED COMPOSABLY TRUSTWORTHY MISSION-ORIENTED RESILIENT CLOUDS (MRC2)	**5a. CONTRACT NUMBER** FA8750-11-C-0249
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62303E
6. AUTHOR(S) Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, Jonathan Anderson, Nirav Dave, Brooks Davis, Jong Hun Han, Steven M. Hand, Alex Horsman, Matt Huxtable, Alexandre Joannou, Anil Madhavapeddy, Theo Markettos, Andrew W. Moore, Alan Mujumdar, Prashanth Mundkur, Robert Norton, Phillip Porras, Colin Rothwell, Charalampos Rotsos, Malte Schwarzkopf, Jonathan Woodruff, Vinod Yegneswaran, Bjoern A. Zeeb	**5d. PROJECT NUMBER** MRCS
	5e. TASK NUMBER RI
	5f. WORK UNIT NUMBER 11
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-050

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 25443
Date Cleared:

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This is the final report for our (MRC) 2 project, culminating a four-year research and development effort that has investigated clean-slate secure networking and security for cloud computing and cloud storage, with emphasis on resilience and trustworthiness. The MRC2 project was a joint effort between SRI International and the University of Cambridge. The project focused on switching, software-defined networking, and application dataflow in datacenters, with a number of subtended efforts – including aligning algorithm and network topology, achieving greater energy efficiency, understanding the concomitant security tradeoffs, exploring multi-scale computing techniques (including work on multi-threaded and multi-core CHERI), and developing capability-based system-oriented application security models. We have extended Cambridge's CIEL distributed computing environment to address security, incorporating the lightweight Mirage OS operating system, and also developed Dios – a distributed operating system. Dios provides robustness as well as security and compartmentalization, and uses properties of CIEL computations to drive resource allocation, protection, and monitoring at the datacenter scale.

15. SUBJECT TERMS

clean-slate secure networking, Software-Defined Networks, trustworthy switches/controllers, dynamic network analysis, datacenter dataflow, multiscale computing, MirageOS, energy efficiency, tradeoffs, security, resilience

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					JUANITA RILEY
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*
U	U	U	SAR	53	**(315) 330-4879**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

1.0 EXECUTIVE SUMMARY

(MRC)2 project was a joint effort between SRI International and the University of Cambridge. The project focused on switching, software-defined networking, and application dataflow in datacenters, with a number of subtended efforts – including aligning algorithm and network topology, achieving greater energy efficiency, understanding the concomitant security tradeoffs, exploring multi-scale computing techniques (including work on multi-threaded and multi-core Capability Hardware Enhanced RISC Instructions (CHERI), and developing capability-based system-oriented application security models.

This report represents a compendium of our progress for the system and network architecture and development plans that have evolved during the project. It also provides much of the reasoning that has taken place. We have extended Cambridge's CIEL distributed computing environment to address security, incorporating the lightweight Mirage OS operating system, and also developed DIOS – a distributed operating system.

DIOS provides robustness as well as security and compartmentalization, and uses properties of CIEL computations to drive resource allocation, protection, and monitoring at the datacenter scale. Under the general rubric of CIEL, Mirage OS, and DIOS, we have developed FABLE, a flow-aware input-output system, which provides an efficient zero-copy data transmission interface that automates the selection of the underlying transport, and the facility to dynamically reconfigure transports as system conditions change. The implications of extending the OS with explicitly I/O flow tracking are significant – eliminating resource contention, upgrading to transparent transport-level security, and increasing robustness via multi-path TCP. FABLE integrally hierarchicalizes the hardware, virtualization, the operating systems, the communication channels, trust, buffer sizes, and higher-level data transformations.

The resilient distributed switch fabric (RDSF) replaces centralized switch infrastructure with a high-dimensional communications fabric, offering potential improvements in security, scalability, energy use, and resilience. CHIMERA is a capability-oriented, rack-scale memory interconnect that will extend SRI and Cambridge's CHERI capability hardware architecture beyond a simple cache-coherent multicore. Trustworthy programmable switch controllers (TPSCs) distribute switch management throughout datacenters and cloud computing. This will offer improvements in security and robustness/resilience, as well as a distributed platform for switch-control applications within the rubric of software-defined networking (SDN). Cloud analysis and misuse detection (CAMD) will enable scaling of existing techniques from assumptions of a single, centralized control points to supporting distributed detection and enforcement within RDSF and TPSC/CAMD SDN environments. We have open-sourced most of the SDN components and have developed numerous interactions within the SDN community.

This is the final report for (MRC)2 project, culminating a four-year research and development effort that has investigated clean-slate secure networking and security for cloud computing and cloud storage, with emphasis on resilience and trustworthiness.

2.0 INTRODUCTION

(MRC)2 investigated key research problems in cloud-computing datacenters, servers, and their networking. These problems include being able to take advantage of multidimensional trade-offs among security, scalability, energy efficiency, and resilience. We built on the foundations of a companion DARPA project, CRASH-worthy Trustworthy Systems Research and Development (CTSRD), which is investigating new hardware, software, and formal methods techniques for host security, under the DARPA Clean-slate Resilient Adaptive Secure Hosts (CRASH) program. Whereas CTSRD limits itself primarily to the confines of single host systems, (MRC)2 extended our work into large datacenters, with particular focus on distributed programming models, network interconnects, and software-defined networking – which enables dynamic reconfiguration of networks to accommodate real-time resource needs and real-time responses to attacks and accidental outages.

Our cloud-computing and networking model combines a number of concepts that have evolved from earlier research in computer systems, distributed systems, networks, and trustworthiness. These concepts include client-server computing and data storage, thin-client systems, high-performance and energy-aware datacenters, and distributed programming frameworks within datacenters as well as between clients and servers. A key aspect of this model is multi-tenancy, datacenter connectivity, storage, switches, and computers are shared by mutually suspicious parties (e.g., each of which is potentially untrusted and/or untrusting of others) as a utility, requiring the application of security measures not only to the client-server interface, but also among applications running in the same datacenter and among multiple switch controllers in large installations. (MRC)2 sought to replace Internet-based datacenter switching and CPU connect technologies with strong local communications primitives that accept multi-tenancy and untrustworthy data as basic precepts, implement mission security policies, and detect and respond to anomalies in network configurations through sound dynamic reconfigurations that offer introspective systemic responses to attacks. (MRC)2 aligned security with the physical topology of datacenter and local network communications, with the goal of improving security, robustness, resilience, performance, and power use.

2.1 Cross-cutting Themes in (MRC)2

(MRC)2 pursued various cross-cutting themes, most importantly:

- Aligning algorithm and network topologies for security, scalability, and resilience
- Understanding and taking advantage of beneficial tradeoffs between security and energy efficiency
- Advancing multi-scale computing techniques
- Exploiting capability-system security models to achieve the foregoing themes

We had originally conceived the (MRC)2 project as a collection of such research themes, with the hopes that we could develop significant synergies among them. As the project progressed, we increasingly found synergies among these themes, and also found that the systematic and principled approach we have taken allowed us to integrate some of the pieces rather easily. This final report for (MRC)2 presents the details of how our efforts have evolved, discusses the extent to which we will have been successful in achieving our goals and in increasing the coherence and consistency of our efforts, and outlines some further work that would be most valuable if pursued along the lines suggested.

2.2 Security, Resilience, Performance, and Energy Use

Security, resilience, performance, and overall energy efficiency are total system-network emergent properties that must be defined hierarchically, because they have potentially different meanings at each layer of abstraction from hardware to low-layer system software to single-domain subnetworks to the totality of connected networks. Furthermore, these properties may be interrelated. For example, resilience ultimately depends on security, reliability, availability, survivability, the dynamic ability to maintain or restore adequate performance in the face of a very wide range of adversities from hardware failures, software bugs, malware, external attacks, insider misuse, power outages, and animals chewing through cables, etc. It is just one more characteristic that must be trustworthy, in the sense that the desired requirements for security, resilience, and guaranteed performance in the face of adversities might be demonstrably satisfiable.

Today's global networking as well as internal datacenter communications are premised on multi-layer, high-performance switches that today typically exhibit some undesirable properties:

- Centralized points of failure
- Inflexible mappings of Internet-inspired architectures to divergent objectives (a) undermining resilience and (b) impacting cost/performance
- Inefficient/disproportionate energy use
- Insufficient attention to security considerations in the logically centralized network control models found in Software Defined Networking approaches
- Ill-specified switch-control models, especially in how switch data paths are affected by control instructions

Our (MRC)2 project addressed all of these and other concerns.

2.3 CTSRD Foundations

To a useful extent, (MRC)2 built itself on CTSRD, CTSRD is a joint SRI and University of Cambridge project on clean-slate host security spanning CPU instruction-set architecture, operating systems, virtual- machine monitors, programming-language compilers that understand the hardware, and judicious use of formal methods – primarily in the analysis of the hardware specifications. Primary elements of CTSRD include the following.

- BERI: Bluespec Experimental RISC Implementation. An open-source platform for research into the hardware-software interface: a multi-threaded 64-bit MIPS instruction-set architecture (ISA) soft core and complete software stack including LLVM and FreeBSD.
- CHERI: Capability Hardware Enhanced RISC Instructions. The CHERI instruction-set architecture provides CPU support for efficient, programmable, and formally grounded software compartmentalization [1]. The CHERI hardware-software system

uses the CHERI ISA to provide a hybrid capability-based system architecture in which capability-oriented programs can execute side by side with conventional software – dramatically reducing the security risks from malware, intentional misuse, and human errors, without interfering with overall system integrity,

- TESLA: Temporally Enhanced System Logic Assertions. Dynamic checking of safety assertions for C-language system software. TESLA could be very useful in dynamic detection of violations of the (MRC)2 resilience requirements.
- SOAAP: Security-Oriented Analysis of Application Programs. SOAAP may have some relevance to the partitioning of the (MRC)2 switch and switch controller functionality.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

Resilience is a global property of a system, built on the careful composition of its parts. Resilience is also a property necessarily specific to the goals of a system in deployment, rather than a static property coming out of design. The (MRC)2 architecture addresses many levels in the software and hardware stack, linked by a programming model that directs local configurations in order to ensure global security, scalability, energy efficiency, and resilience properties. Mapping application structure into local enforcement is a key aspect to the (MRC)2 approach, as awareness of application requirements in the context of a mission will drive the investment of a variety of resources into supporting or enforcing required properties. A key driving force is proportionality: greater investment of resources that will result in stronger properties, such as security, performance, or robustness.

The technical components of (MRC)2 are rather diverse. The components are arbitrarily more or less according to our original proposal into the following categories listed below:

The above list reflects the emerging elements/components, but not their structure and interrelationships – which are still evolving. Figure 1 illustrates how these key technologies relate to overall datacenter architecture.

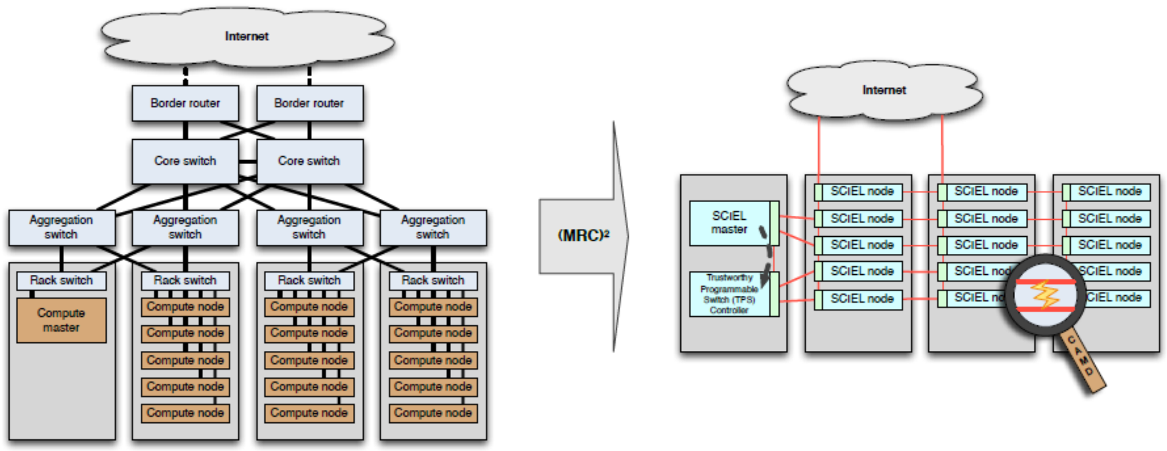


Figure 1: MRC2 Transforms Datacenter Architecture

3.1 Extending CIEL with MirageOS and DIOS

The goal is to develop a heterogeneous trustworthy distributed programming framework for datacenters. This framework is based on CIEL, Cambridge's distributed heterogeneous programming framework, with transparent support for distributed scheduling, fault tolerance, and consensus gathering. CIEL employs cryptographic hashes in naming computation stages, relying on idempotence to allow computations to be restarted and replicated for robustness.

The robustness properties and security properties of CIEL form the basis for (MRC)2 upgrading it by mapping CIEL computation topologies into CTSRD and MRC2 technologies for containment and resilience. Two operating-system components contribute to robustness and security: MirageOS is a lightweight unikernel operating system based on OCaml. DIOS implements a distributed trusted computing base (TCB) for data-flow computation, employing a blend of local and distributed enforcement unavailable in current switching fabrics. DIOS also extends beyond the OS kernel to take advantage of hardware virtualization primitives to provide a secure, type-safe coordination layer that sets up computation inside secure containers with controlled inputs and output channels. The interrelationships among these three components are shown in Figure 2.

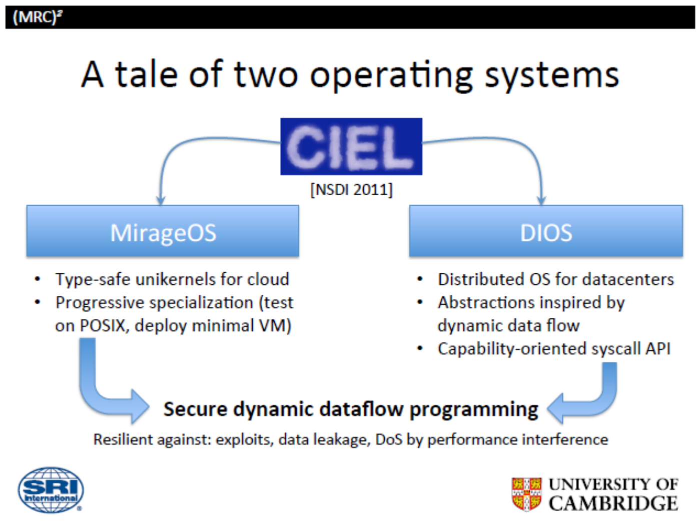

(MRC)²
A tale of two operating systems
CIEL
[NSDI 2011]
MirageOS
DIOS
Type-safe unikernels for cloud
Progressive specialization (test on POSIX, deploy minimal VM)
Distributed OS for datacenters
Abstractions inspired by dynamic data flow
Capability-oriented syscall API
Secure dynamic dataflow programming
Resilient against: exploits, data leakage, DoS by performance interference
SRI International
UNIVERSITY OF CAMBRIDGE

Figure 2: Secure Dynamic Data Flow Programming

3.2 Arguments for (MRC)2 Trustworthiness

(MRC)2 considers cloud resilience to be one of the primary attributes of overall trustworthiness, co-equal with other system, network, and cloud-resource attributes such as security, integrity, survivability, and reliability. Thus, we generalize our response to represent what (MRC)2 intends to do for trustworthiness, albeit including specific references to resilience. It is our fundamental belief that resilience cannot be achieved without adequate trustworthiness with respect to certain other attributes. In particular, considerable trustworthiness, with respect to resilience and related attributes, can be expected to result from our system/network/cloud architectures and their carefully structured implementations from the predictably composable modularity of the switchlets, switch controllers, energy-efficient datacenters, and from our assurance techniques applied to the hardware and software involved.

Our approach to assurance makes judicious application of formal methods where most effective, as one way of dramatically increasing assurance — in addition to the more conventional methods of prototype development, testing, and red-teaming. In part, we are relying on our CTSRD CRASH project to provide some formal analyses of the underlying CHERI hardware specifications. We also hope to be able to model certain properties relating to resilience, reliability, security, integrity, etc. — in clouds, switches, servers, and other relevant components – as funding and time permit. We also seek to model some of the most critical behavioral aspects of administrators and users, to the extent that they are fundamental to achieving adequate trustworthiness. The resulting formal analyses would be very useful in detecting design flaws, implementation errors in hardware and software, and operational issues relating to security, efficiency, and usability in networking and cloud systems.

The (MRC)2 design is intended to support the building of trustworthy datacenter networks as well as trustworthy software-defined networking (SDN). Each technical element/component (CIEL, MirageOS, DIOS, FABLE, RDSF, CHIMERA, TPSC, and CAMD) contributes to an overall argument for trustworthiness by supporting the mapping of datacenter-scale computation goals into underlying computation and communication primitives in such a way that mission properties are, to the greatest extent possible, maintained. Of course, it is the composition of all these components that is particularly important in assuring the trustworthiness of the emergent properties that arise from these compositions. Unlike previous efforts, (MRC)2 allows explicit reasoning about tradeoffs between security, scalability, sound dynamic reconfigurability, energy use, and resilience, with annotations at the programming and management layers to drive investment of conserved resources.

3.2.1 Mirage OS

The initial prototype of MirageOS is built on top of the Xen hypervisor [2] with Virtual Machines providing isolation between processes, but future versions will support the CHERI and Capsicum capability systems to provide isolation guarantees on different architectures. The DIOS coordination layer must be as minimal and safe as possible, and so minimize unnecessary components. Conventional virtual appliances (e.g., web servers) are similarly built to provide a small, fixed set of services. However, the VM image contains a number of components that are rather loosely coupled: a guest OS kernel and user space binaries that typically attach an external storage device with configuration files and data. Thus, even the simplest appliance VM contains several hundred thousand, if not millions, of lines of active code that must be executed every time it boots. Much of this code is due to a need for backwards compatibility with existing applications, such as the POSIX API for processes to interact with their environment. There are no standards for many aspects of application configuration, and so Linux distributions typically resort to extensive shell scripting to glue packages together.

Mirage: An OCaml Library OS A libOS is structured very differently from a conventional monolithic OS. All services, from the scheduler, to the device drivers, to the network stack, are implemented as standalone libraries that can be linked directly with the application. A consequence of this is that applications can configure services programmatically by directly invoking the library calls, instead of calling across a different protection domain - as with a conventional kernel/user space. We explore a new sort of libOS, one built directly in a type-safe language, with applications taking advantage of the extra semantic information exposed in higher-level interfaces than those exposed by C. For our prototype unikernel implementation, we use the OCaml runtime running on the Xen [2] hypervisor. Notice that we deemphasize strict backwards compatibility with existing applications at the source code level, and instead support it at the network protocol level. Existing code can easily be run in separate VMs due to our use of virtualization.

OCaml is a modern functional language supporting a variety of programming styles, including functional, imperative, and object-oriented. It is a dialect of the ML family, with a well-designed, theoretically sound type system that has been developed since the 1970s. ML is a pragmatic system that strikes a balance between imperative languages, e.g., C, and pure functional languages, e.g., Haskell. It features type inference, algebraic data types, and higher-order functions, but also permits references and mutable data structures while guaranteeing that all such side effects will never cause memory corruption. Safety is achieved by a combination of compile-time type checking and dynamic bounds checking of array and buffers. The compiler supports a portable bytecode and several native code targets (x86, ARM, PPC) as well as more exotic runtime targets such as 8-bit PICs and JavaScript. OCaml was also a pragmatic choice in which to implement Mirage as it is the implementation language for the open-source Xen Cloud Platform [3] and critical system components [4, 5]. On the other hand, using OCaml necessitated a significant engineering effort to rebuild many standard system components, particularly the storage and networking (e.g., TCP/IP) stacks. Mirage links OCaml code into kernels that run directly on a Xen hypervisor. Our design minimizes runtime complexity, preferring implementation of all but the lowest-level features in a safe high-level language. We now discuss some of our core design decisions:

- *Parallel Protection Domains:* Unikernels link an application and language runtime into a uniprocessor VM that has a single 64-bit address space. Parallelism is obtained by running multiple VMs and message passing between them, as with the Barrelfish multikernel [6]. These VMs need not run on the same physical host, although communication is more efficient if they do.
- *Protocol-level Compatibility:* Cloud services mostly use Internet protocols to communicate between services – e.g., via HTTP as an RPC mechanism. Mirage unikernels communicate externally via these protocols, while internally eliminating binary interfaces where possible via static link time optimizations.
- *No Dynamic Loading:* Mirage unikernels are partially evaluated during compilation, e.g., to incorporate static configuration files, and sealed [7] at runtime to prevent self-modifying and dynamic loading of code. Appliances are reconfigured by compiling a new image, eliminating the complexity of dynamic code and permitting additional compile-time optimizations.
- *Statically Typed Libraries:* All system services are type-safe, re-entrant libraries, and range from the protocol-level (HTTP, SSH) to networking and storage (TCP/IP, FAT32) to the core library (threading, binary stream manipulation). Data copying within the stack is minimal and buffers are fully bounds checked.
- *Cooperative Concurrency:* Appliances are hardened against external network attacks via type- safe I/O, but code within the appliance is trusted. Lightweight threads cooperatively decide their yield points, similar to Nemesis's [8] provision of application-level quality of service.

The most specialized output of the Mirage compiler is a unikernel, a standalone kernel with a minimal OS runtime that uses the hypervisor interfaces directly. It consists of the PVBoot library for initializing a basic computation environment, a modified language runtime library for heap management and concurrency, and type-safe device drivers that interface with the external world via the safe I/O stack. Finally, since unikernels are single-address space, they can be sealed to significantly improve their security against various threats.

3.2.2 DIOS: Secure Distributed Operating System

DIOS is built for scalable, transparent distribution of operations across many nodes in a datacenter. DIOS is a special purpose operating system for Warehouse-Scale Computers (WSCs). In this endeavor, it is part of a substantial lineage of past research on distributed operating systems [9, 10, 11, and 12] – but yet, DIOS is different. It approaches the distributed OS concept with the hindsight of modern distributed systems theory and applications. DIOS reflects three key themes and exposes the necessary abstractions to offer this functionality to user applications. The three key themes are:

1. Naming and locating system objects (I/O targets, devices and programs),
2. Allocating and managing hardware resources, virtualizing them where necessary, and
3. Effecting privileged operations on the behalf of user applications, while isolating them from each other.

DIOS, since it is a distributed operating system, offers functionality across multiple nodes in WSC that coordinates nodes reliably and deals with the inevitable faults. Furthermore, it is able to do so at the scale of hundreds or thousands of nodes. Unlike traditional OSes, DIOS integrates inter-machine communication and state maintenance in the privileged OS kernel. This offers the operating system more information to work with than it would normally have available. Let us compare with a scenario where all information pertaining to distributed operation is stored in the user application, and all privileged information pertaining to the local application process is stored in the kernel. Yet, despite this expanded role of the OS, users should not find it overtly difficult to program the system – despite the inherent complexity of the distributed operating environment. It must be possible to write working programs against simple, transparent abstractions and have sufficient information that must also be exposed to the user to enable optimize applications.

While the OS is aware of distribution, DIOS leaves higher-level policy decisions – such as where to locate data in the distributed system, or whether to maintain strongly consistent replicas – to user-space applications, rather than encoding them within its abstractions.
Security and isolation are key concerns for operating systems and distributed systems environments alike. DIOS must not only be able to guarantee the same level of isolation and protection as a traditional operating system, but indeed needs to take it further and guarantee isolation of across multiple nodes too. Furthermore, it should mandatorily track information flow in the distributed system, as well as offering a practical way to restrict exposure of data. Again, this must work reliably across machines and at scale.

Finally, it is unreasonable to expect the world to change overnight and for DIOS to offer sufficient benefit to motivate re-writing all WSC software. Hence, an incremental migration to running increasingly large portions of a WSCs workload within DIOS must be feasible. While not all of the benefits of DIOS might initially be attainable, each successive migration step ought to offer further improvement over the previous state. All of the concepts described above contributed to shaping the high-level DIOS design principles.

In summary, they are as follows:

- Expose scalable, transparent abstractions that support both local and distributed operation.
- Enable application-level policy decisions by mandating minimal mechanism in the operating system abstractions.
- Mediate WSC-wide information-flow through a distributed capability delegation model, enabling selective exposure, information flow control and data provenance tracking.
- Offer an opportunity to incrementally migrate to the new abstractions and combine them with legacy application code.

3.2.3 FABLE: Flow-aware Input-output System

FABLE consists of a user-space application library, an extra system call to register with a new name service daemon, and some extensions to existing polling system calls to support the new I/O descriptors. Figure 3 illustrates the following stages of FABLE session:

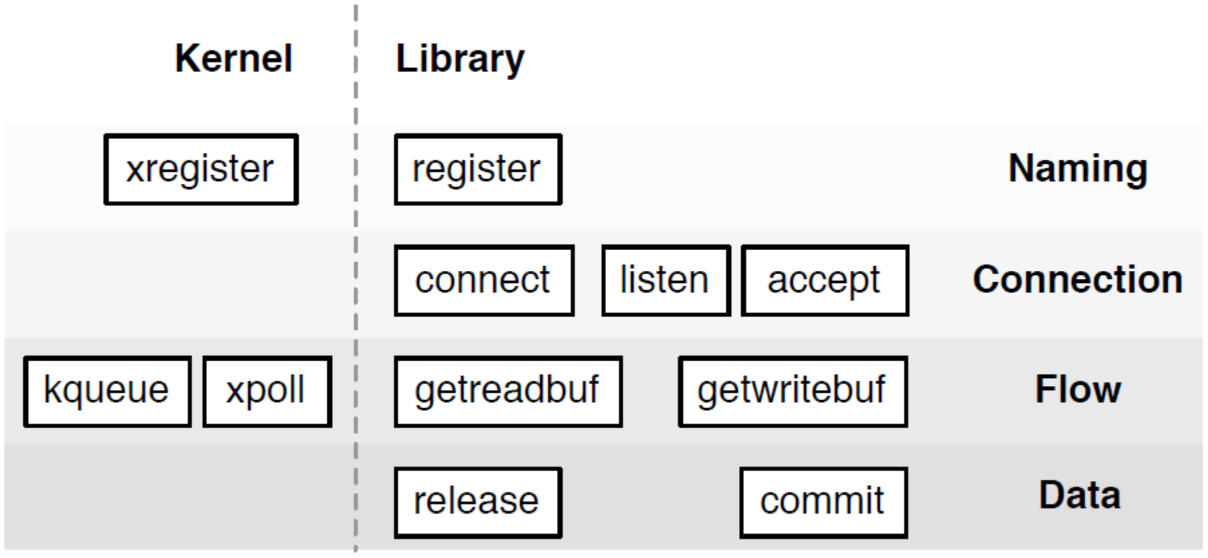


Figure 3: Stages of a FABLE Session

- *Naming*: All end points are explicitly named, and a system service (opaque to the library user) tracks the location of processes and virtual machines and notifies them of reconfiguration events. If virtualized or running in a cluster, this name service can register with a higher-level service that has more accurate system-wide knowledge.

- *Connection*: Connection setup is similar to POSIX sockets, except that the end points are named services. Every connection has a single transport mechanism, ranging from tightly coupled shared memory, to a TCP connection, to a page-flipping memory pipe. The client specifies if the remote end point is trusted to cooperate, or if private data copies are required.

- *Flow*: buffered structures for reading and writing are always allocated by the FABLE library, and are tailored for the connection in which they are associated with (e.g., an entry in a shared memory ring). Buffers are single-use only, and buffer creation calls are where back pressure is applied, rather than at the point of reading or writing. If buffers are unavailable, the application polls to be notified when more are available.

- *Data*: every buffer is owned by exactly one FABLE connection, and ownership is transferred either via a release back to the system (e.g., after a read), or via a commit to write it onward to the next end point. Once ownership has been transferred, it can never be regained and new buffers must be requested.

When designing FABLE, we assumed that nested scheduling layers will dominate architectures for some years, due to the popularity of virtualization and multi-core hardware. The addition of a system-wide name service for end points is key to keeping track of I/O flows in such complex environments. The FABLE name service is hierarchical and can keep higher-level software layers informed about activity within a particular domain, up to and including a distributed cluster of physical hosts. The ultimate goal is to form an accurate view of dataflow requirements for all applications across a cluster, and provide more structured information for schedulers to maximize I/O throughput across a cluster.

The FABLE name service is used only to coordinate the establishment of data channels and track their lifetime. Once established, the data transfer between two end points is designed to be highly efficient and not require a system call for a read/write operation, although some transports may choose to do so (e.g., remote TCP when using kernel sockets). This is particularly important for throughput in virtualized environments, where system calls can be disproportionately expensive due to privilege checks by the hypervisor.

Although FABLE provides a new API, it can also be integrated directly into existing applications via a socket compatibility layer. As we noted earlier, the sockets layer forces at least a single data copy and so is often less efficient, but the facility to track all I/O operations across the system remains extremely useful.

Every FABLE connection is associated with two named end points. The application calls the `xio register name` to register a new end point, and obtains an opaque `xio context` structure in return. The library does not keep much state—instead, it accesses a system-wide name daemon via a kernel file-descriptor interface and uses this to register with the name service. This descriptor is used by the name service to track the `xio context` for its lifetime, including the details of where it is scheduled, and the connections emerging from it.

Most of the policy behind connection handling is implemented in a user-space daemon that listens for FABLE registrations and scheduling changes from the kernel. This daemon is responsible for implementing all the policy for connection rendezvous between end points, and acts as a system-wide database of I/O flows. When running on a native kernel on a physical host, it is primarily concerned with ensuring that communicating processes are scheduled close to each other (from a NUMA and core layout perspective). However, once virtualized, it registers with the VM management stack and keeps it informed of the event stream. Similarly, if a host joins a cluster of physical machines and wishes to cooperate with them, the name service can integrate with Zookeeper [13] to handle distributing its local metadata to the other hosts.

Once `xio_context` has been obtained, it can be used to establish multiple connections to other end points via `xio_connect`, and also to listen for incoming connections via `xio_listen`. FABLE names are URIs and so the connection API converts a name into a concrete connection, with the application unaware of the precise transport unless it has been explicitly specified in the name. The `xio` schema is reserved for FABLE-aware end points, and some other schema such as `tcp` or `udp` are supported to facilitate external communication via standard protocols and are needed for the socket emulation library.

Connection
Connection establishment requires both end points to agree that they wish to communicate (i.e., that one is in a `listen` mode and the other is connecting), and the selection of a transport mechanism that is agreeable to both ends. Since the FABLE name service has both of the services registered, it acts as the intermediary and calculates the best transport protocol for the two end points. A successful `xio_connect` library call will return an opaque `xio_handle` that is used to reference the connection by the application.

The details of transport selection are necessarily quite complex, since they depend on some static factors (hardware memory and core layout) and dynamic factors (e.g., virtualization introducing external load). The system name service is thus better placed to make this decision, instead of the application itself.

Data Transmission
Applications never allocate their own I/O buffers, and instead obtain buffers using the `xio_getreadbuf` and `xio_getwritebuf` calls. This allows FABLE to allocate optimal buffers for the transport associated with the connection—e.g., low memory if the network card requires it for DMA, or from a shared memory segment on the closest NUMA node, or with space reserved for TCP/IP packet headers. These are optimizations reminiscent of exokernels [14] and explicit path selection [15] that have so far not found their way into mainstream UNIX-like systems. Buffers are very similar to iovec structures, and include a pointer to the I/O memory and its size. They also include a reference to the xio handle that created them, and an epoch number to help with reconfiguration. An important property of buffers is that they are single use, and cannot be reused once they are freed or written to another end-point.

An `xio_getreadbuf` call is non-blocking, and returns an array of buffers that are filled with data, or an empty set to indicate that the application should poll for more data. When an application is finished with a buffer, it must call `xio_release` to hand it back to the system. Since there is a limited set of buffers associated with each connection, the `xio_getreadbuf` call can return an `ENOSPACE` to indicate that the application is holding onto too many read buffers and should release some before requesting more. To prevent deadlock, the application may handle this by copying read buffers into private memory and releasing them early.

The write data path calls `xio_getwritebuf` with an optional parameter to specify the maximum size of the available data. This returns an array of buffers that should be filled in any order by the application. The size of each individual buffer is very transport-specific, and ranges from a 4K page size for shared memory, to slightly smaller than an interface's MTU for TCP to reserve space for packet headers. When a buffer is filled for writing, the `xio_commit` call will transfer ownership of the buffer back to the library, which queues it for writing. The application may no longer modify or access this data once it has been committed—this is advisory if the connection is trusted, and otherwise enforced via a private copy being taken by the receiver or the page reference being unmapped from the transmitting end.

The notion of single ownership of buffers is key for constructing efficient stream processing engines, where processes perform a combination of data processing and proxying. For example, consider a web server that reads pages from disk via one FABLE channel, and transmits the disk pages to a `memcached` process, which then serves it to a network interface. The connection from the disk layer will be a set of page-aligned buffers, whereas the connection to the memory cache is a large shared memory ring. In this situation, the application may commit a read buffer from the disk channel *directly* into the `memcached` channel, despite the disk buffer not being obtained from the memory cache channel. Every buffer tracks its home connection, and so the FABLE library performs the appropriate translation to convert between transport mechanisms (usually via a slow copy). Once the foreign buffer has been committed then the upstream writer is responsible for releasing it.

Reconfiguration
Every `xio_handle` also has a file descriptor that can be obtained to poll for reconfiguration events. A reconfiguration indicates that the underlying transport mechanism is being changed, and that the application should drain any older buffers as quickly as it can. This is accomplished either by releasing them, or committing them for a write. The *epoch number* in each buffer is used to distinguish between the different transport mechanisms see Figure 4.

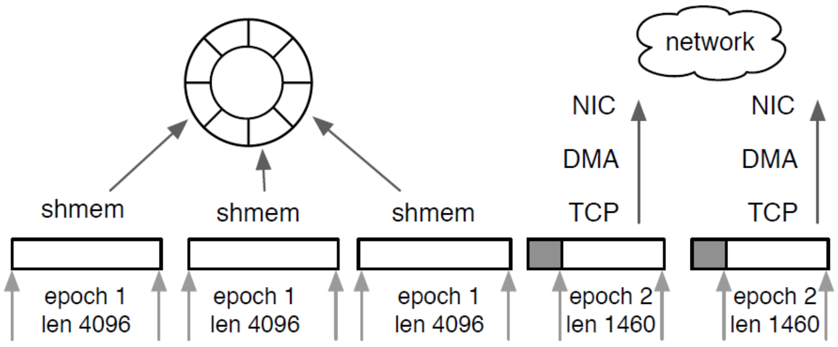


Figure 4: FABLE Buffers Shared Memory Connection

While the reconfiguration notice to the application is synchronous, the actual change is very asynchronous and similar to Xen live migration [16]. The new transport data path is established first, without altering the existing one. A notification is then sent to the FABLE library instance via the event file descriptor associated with the end-point. All new buffers requested by the application are now associated with the new transport, and the application is given a timeout period to use the old buffers. If the application fails to drain them in time, FABLE can slowly proxy the old buffers to the new transport if it can, or simply terminate the connection.

A good example of the need for reconfiguration is when using a virtual machine cluster. When two VMs are on the same physical host, they establish an inter-domain shared memory communication (e.g., via *libvchan* in recent versions of Xen). If one of the VMs then live migrates to a different physical host, this connection would normally be terminated. With FABLE, however, the live relocation is observed by the cluster name service and triggers a recalculation of the transport protocol, and configures TCP instead. The example flow of buffers can be seen in Figure 4.

Aside from this, many of the performance anomalies we observed earlier can be adjusted for via a reconfiguration process. For instance, if two cores are idle and not virtualized, then a low-latency spinning transport may be the most efficient. If another end-point is subsequently scheduled onto the same cores, they will begin contending, and the transport should be reconfigured to a futex-based version. Similarly, if a process is rescheduled to a different NUMA node, this can trigger the reallocation of memory buffers to ones from the new NUMA node.

There is some resource cost associated with reconfiguring a channel, and it is not intended to be done extremely regularly. Instead, the FABLE name service observes all I/O flows on the system and can be configured to either automatically balance them (e.g., using Kalman filters to smooth out changes [17]) or allow a system administrator to optimize it manually if desired. Either policy is easy to implement due to the existence of the system name service to aggregate and coordinate any reconfigurations.

3.2.4 RDSF: Resilient Distributed Switching Fabric

RDSF could be considered a robust datacenter network infrastructure. However, alongside quantifiable provision of robustness and resilience along with traditional metrics of throughput and latency, an infrared image of heat dissipation within a datacenter. In traditional datacenter design, per-rack switching is rarely an energy-scalable commodity: operating fully, or not at all — every 10GbE port illuminated, the entire switch fabric operating continuously: poised to move data at full capacity even if the utilization is only housekeeping, even if the host systems are powered off.

Our architectural approach, Figure 1, distributes the switching fabric across the datacenter; the fabric is distributed at the granularity of host: one switch per host. We acknowledge that this might not be the ideal granularity for every task. However, a one-to-one mapping provides the most-ideal setup for work focused upon resilience and robustness as well as providing high levels of path-programmability between elements. Alongside this, the energy dividend of this structure is that we have a highly granular scalable switch and host structure able to be dynamically powered on and off to suite demand.

The distributed switch fabric permits us to focus on providing a workload-proportional energy-consumption model. The issue of proportional energy use, elegantly espoused previously [18], has largely focused upon efficient host use. High-radix connectivity channels provide some of the advantages of past architectural proposals – e.g., [19] – but through implementation in hardware we achieve more flexible, arguable more robust and notably considerably more capable implementations.

Our approach is a high-radix, multiple-dimension switch fabric interconnecting each host within the datacenter. This is achieved using the programmable switch fabric provided by the NetFPGA 10G platform. Work on the NetFPGA infrastructure has led to a high-speed switch system that is able to provision a multiport 10GbE interface OpenFlow, SDN switch. The NetFPGA has a natural connectivity for four 10GbE ports and two further high-speed presentations, each capable of 65Gbps, along with a local PCIe capable of operation at 32Gbps. Through the use of cut-through packet passage it is plausible to assemble a low-latency switch fabric that via a six-port-per-card installation permits our high-radix, hypercube-like, structure, all while leaving sufficient local resources to provide the required level of programmability in end-host switches and the necessary intermediate switching stages. This programmability has included the co-implementation of an OpenFlow switch and CHERI processor. There has been effort to port the CHERI processor to the NetFPGA platform – opening up the opportunity for the hardware-based capability enhancements offered to operating systems, with applications (e.g., CAMD) also being made available in the network- control context. SDN components such as the implementations of the OpenFlow switch interface software and OpenFlow controllers, core to the current Software Defined Networking efforts as well as being core to our solution, would greatly benefit from the offerings of a capability-enabled approach. Finally, we have a ready-made solution if the combination of CHERI and the performant OpenFlow switch is too much for the NetFPGA hardware – namely, to use the NetFPGA as a switch-enabled host adapter interfaced with a current CHERI instance on the current DE4-based systems. Such a configuration, shown in Figure 5, hybridizes the two systems by connecting along a well-defined interface (PCI-Express).

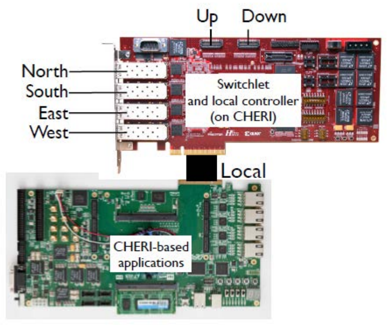


Figure 5: Hybrid switch and CHERI using NetFPGA10G and DE4 hardware

3.2.5 Chimera: Capability-oriented, Rack-scale Memory Interconnect

Datacenters should be constructed from units each capable of computation, communication, and storage. This obviates the need for dedicated network switches and unifies the trust model for heterogeneous systems. We believe that system-on-chip and die stacking techniques, already prevalent in the embedded systems space, can be used to provide power-efficient computers at many different scales, from portable in-the-field units to warehouse-scale systems, see Figure 6.

We also observe that silicon photonics and photonic printed circuit boards (PCBs) are advancing, and will offer power savings and facilitate more complex topologies [20].

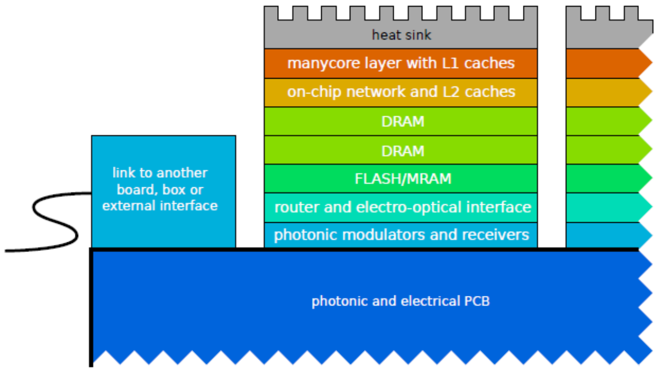

heat sink
manycore layer with L1 caches
on-chip network and L2 caches
DRAM
DRAM
FLASH/MRAM
router and electro-optical interface
photonic modulators and receivers
link to another board, box or external interface
photonic and electrical PCB

Figure 6: Hardware Concept

Current datacenters based around PC class components commonly employ a hierarchy of network switch and routing equipment with computing nodes at the leafs. These networks employ high-cost, special-purpose devices often recreating variations of classical Ethernet and IP-based topologies overlaid onto the physical structure of the datacenter environment. Common structures incorporate an aggregation rack per switch (the top-of-rack switch), and one or more room-wide aggregation switches that provide rack interconnectivity as a star of star networks; variations on this principle abound but the fundamental basis — the use of common switching and routing equipment — leads to the use of power-hungry devices ill-suited to the low-latency communications between nodes in close physical proximity (e.g., adjacent racks). Selecting to enforce a strict star hierarchy upon the network infrastructure has other disadvantages. It brings with it only weak mapping between the optimal data structures and the best communications structures within the datacenter in latency and in colocation of task with data, but it provides a weak fault tolerance model.

On the other hand, at a low level our heterogeneous computer systems already use PC-area high-speed serial communication mechanisms to talk to GPUs, NICs, and disk controllers. Currently, these low-level communication mechanisms mimic old bus-based protocols: violating trust models (or even simple virtual memory protection) and exhibiting little redundancy. From an electronics perspective, there is no barrier to making these links form a communications fabric capable of both low-latency and resilient communication.

We are using commodity field programmable gate arrays (FPGAs) to prototype these systems since they have the required high-speed communication links and can support substantial computer systems. Such infrastructure can be used to not only implement systems but also to emulate behavior and monitor performance (e.g., predict power consumed). A highly distributed communications structure has previously been limited due to management complexity. Yet, an advantage of a software-defined network such as the approach of OpenFlow permits the orchestration of the entire switch fabric. Thus, the FRESCO framework described earlier can present and manage the datacenter as a contiguous unit, subdivided into virtual subsystems as required – thereby permitting unification in policy balanced with the architectural distribution that provides features such as resilience and performance. Further, the FRESCO framework logically provides the perspective needed for optimal cloud analysis and misuse detection.

Current Framework

Both the CHERI and CHERI-2 processors have already been synthesized on FPGAs. Listed below are some of the specifics of the CHERI/FreeBSD layout on the DE4 FPGA:

- Each DE4 board is equipped with two DRAM slots. The single-core version of the CHERI processor uses one of the DRAM channels. In the dual processor system, each CHERI could have access to its own DRAM channel. The L2 cache can also be allowed a dual-port access to the DRAMs.

- A significant part of the LUTs on the FPGA are used for dealing with the caches. The caches themselves are located on dedicated BRAMs within the FPGA fabric. Enough space is available on the BRAMs to accommodate the caches. In modern processors, the caches often occupy 30% or more die area. In some processors, the numbers are as high as 50%.

- Overall, CHERI occupies around 40% of all the FPGA resources. An accurate figure cannot be given at this point, as the processor is still under development. Assuming a stable allocation of resources, the full chip could accommodate at least 2 CHERIs.

- The SD card on the DE4 stores the boot image for the processor. FreeBSD can be loaded onto the SD-card and booted. The card could be loaded with a special multi-core boot procedure that will allow a dual-core CHERI initialization.

Communication between multiple FPGAs is currently done through the Reliable Link layer developed by Simon Moore. This layer guarantees low-latency bidirectional communication between the FPGAs. We plan to add another layer on top of the Reliable Link layer, which will deal with cache coherence between the FPGAs. This abstraction layer needs to be intelligent in order to cope with potential failures in the inter-FPGA links. In CMPs, the processor cores and NOC are implemented on chip; thus, failures in the NOC are highly unlikely. The Bluehive cannot guarantee such reliability as the links are external to the FPGA and prone to interferences. The system will require some redundancy in order to cope with failures. Techniques such as adaptive incremental check pointing can be used to deal with failures [21]. This technique allows periodic snapshots of the current state. If an error occurs, the system can be rolled back to the last known state.

As the design will be implemented in hardware in the Bluehive, we intend to use the existing link topology. The current topology is a 3D torus. It allows fast bidirectional communication between six neighboring FPGAs, four over SATA3 links and the other two over eSATA. As the system has very low latency the penalties for seeking farther FPGAs is not high when compared to typical DRAM access times. The overall design of the system will appear as a large tiled CMP. The operating system running on the proposed architecture will perceive the system as a single, albeit very large, multi-core processor. The link mechanism should give an illusion of shared memory through the abstraction layer. DE4s are equipped with four 1G Ethernet ports. Following the system shown in Figure 7, one of the FPGA could be declared as the master and given access to the Internet via a TCP/IP layer.

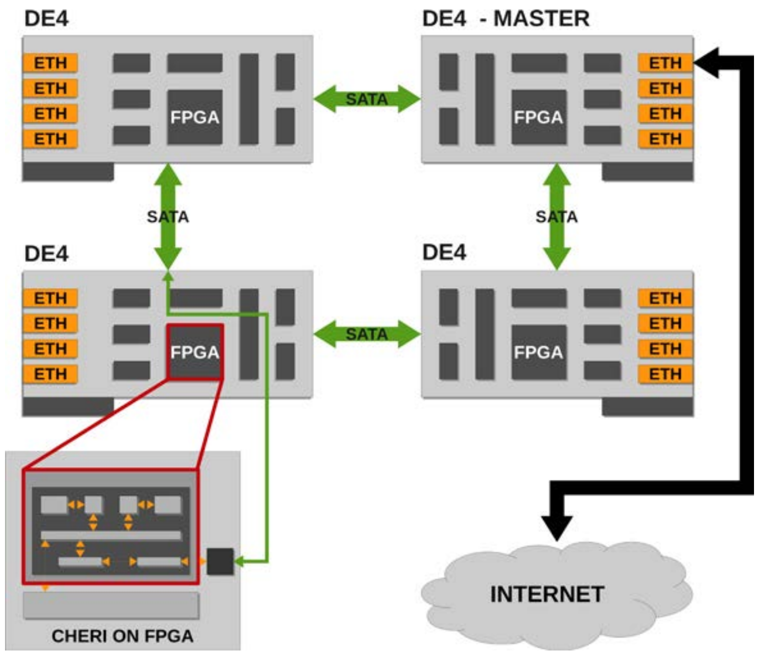


Figure 7: Connecting a multi-core CHERI system to the Internet

When coupled with capability functions, the CHERI processor adds a capability cache that adds more circuit logic to the design.

Work done by Robert Norton in his PhD thesis proposal suggests an alternate mechanism for inter-processor communication, known as remote-store. This mechanism could be further extended with capabilities in order to provide secure communication.

The proposed design has several challenges:

- The modular nature of the system will make it flexible to adapt to varying topologies and core numbers. The system should be capable of identifying any changes to the structure and adapt accordingly. We propose a special boot procedure that will be used to identify the topology, core numbers, and link failures within the system. Checkpoint schemes are often used in supercomputers to deal with failures. As the number of cores, caches, DRAMs, and so on in a supercomputer is large, the mean time between failures is high. Some techniques used in supercomputers – such as the Blue Gene/Q [22] – could be employed in our system. At boot, the cores will check their individual processor IDs (supported by MIPS). They will then communicate with their neighboring processors through the abstraction layer and build a table of all the cores in the system. The list of cores will be used by the abstraction layer to maintain cache coherence.

- Datacenters often suffer from inefficient load balancing, in some cases the load on the edges might be 50% lower than optimal. One such example is the modular shipping container-based datacenter [23]. Performance issues will arise when a single task is spread between several distant processors. Even though the Reliable link layer provides low latency and high bandwidth, the overall effects of heavy inter-FPGA communication could be a potential bottleneck. To deal with this issue, the abstraction layer will serve a dual function. It will have a mechanism for packaging cache lines into a viable format for transportation through the Reliable link. In addition, each pair of links on the FPGA will act as a network router (6 pairs in total when using the PCIe SATA3 expansion card). The router will maintain a routing table that will be populated at boot time. As the network will not change after boot (unless there is a link failure), static addressing can be used. We intend to use the processor ID as the routing address. A distance-vector routing protocol will be necessary for efficient load balancing. Best links will be chosen for the inter-core communication. Several good examples have been shown in [24]. Applications running on this system will be forced to utilize the spatial locality of the cores. The addressing schemes described in [25, 26, 27, 28] could be applicable to this system. As the cache lines used by CHERI and CHERI-2 are 256 bits long (the line sizes could be increased if necessary), a 32-bit address will not be a major overhead for the communication. Other schemes derived from transactional memory can also be used to reduce the addressing overheads.

- The operating system needs to be able to cope with such a dynamic design. The design will be dynamic in the sense that there will be no specific architectural constraints such as a total number of cores, links, and routers. This will all be determined at boot time. Hence the OS will need to be tweaked in the way that it could dynamically adapt to the architecture.

3.2.6 TPSC: Trustworthy Programmable Switch Controllers

Within each logical layer in a network or subnetwork, we anticipate the opportunity to express a variety of security relationships, including asymmetric and mutual distrust, as well as relating to restricted and permitted information flows. The network architectures need to be tailored to the specific uses, which may differ widely between datacenters and general interdomain networking. Thus, a variety of single-switch and multi-switch controllers, as well as multi-controller switches is likely to be desirable. However, note that multiple switches and multiple controllers present some very challenging control problems with respect to consistency, security, and resilience.

The high-level operation of an OpenFlow switch is quite straightforward. As packets stream in, the switch aggregates header information and compares it against the flow table entries. The switch then updates the packet according to the actions prescribed in the matching entry, and sends the updated packet to the appropriate egresses via a crossbar.

Our switch architecture depicted in Figure 8, based upon the reference design provided by the NetFPGA-10G project, is heavily inspired by Yabe's [29] 10G 4-port OpenFlow switch. Our switch parameterized by N, the number of MAC ports in the switch, and W, the width of the internal data plane. These values can be freely changed to meet resource and performance requirements.

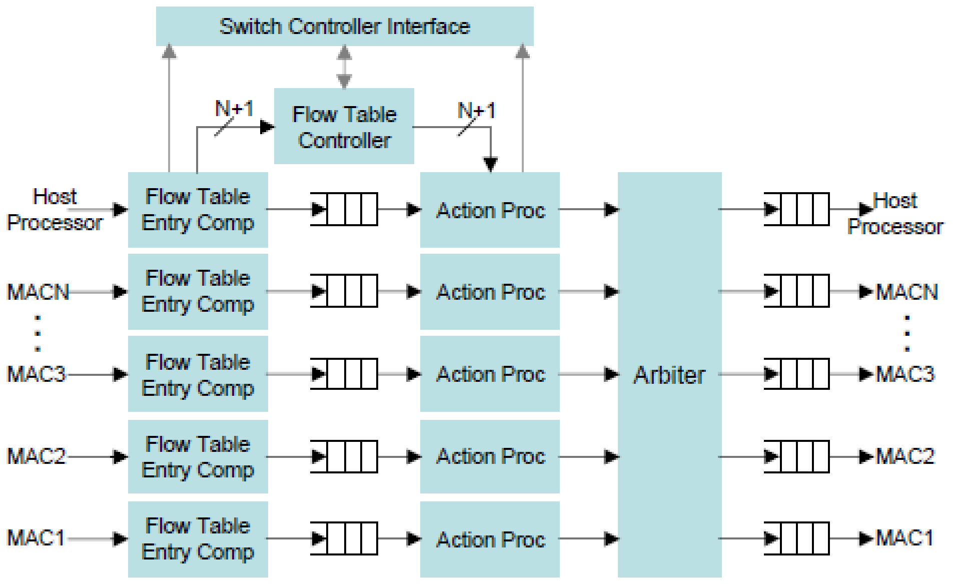


Figure 8: OpenFlow Switch Architecture

In an N-port instance of the switch, there are $N + 1$ ports with the $(N + 1)^{th}$ port reserved for communication with a host processor. The switch also has a controller interface to allow the host processor access to the flow tables and the various registers that maintain statistics.

As a switch is always permitted to drop packets due to over-subscription, it is common practice in RTL switch designs for the data path to be a synchronous pipeline with no back pressure. This reduces some design complexity and eliminates some logic statically. However, this minor efficiency comes at the cost of less-understandable compositional semantics for components and a reduced ability to debug the design. For these reasons, we implemented all the switch component interfaces to provide back pressure via the BSV's ready-enable micro-interface protocol to stall operations when sufficient buffering is not available. This change has a negligible area cost, but it dramatically reduces the design effort. Additionally, all internal switch interfaces follow the standard split-transaction protocol, facilitating latency-insensitive modular design.

The design endorses the "fail-early" principle, dropping any packet for which it cannot guarantee end-to-end buffering. When the header flit of a packet arrives at the switch and sufficient buffering is not available, the header flit and all the subsequent flits belonging to the packet are discarded, and a failure is recorded.

The design of the switch pipeline has been divided into the following modules.

Flow Table Entry Composer: Each input port of the switch receives packets as a sequence of fixed-size flits. For each input port, there is an associated flow table entry composer that aggregates the packet header and decodes it into an internal flow table entry tag representation. This entry is forwarded to the flow table controller as a query. The composer also forwards the entire packet to the corresponding action processor.

Flow Table Controller: The Flow Table Controller is responsible for maintaining the flow tables entries and the per-flow statistics, and arbitrating the requests to access the flow tables. It is implemented as a 10-stage pipeline, and can receive as many as $N + 2$ queries in each cycle: one from the switch controller interface, and $N + 1$ from the flow table entry composers. For every cycle, the controller (via a configurable priority scheme) selects a request and pushes it into the pipeline. It handles requests serially due to resource constraints. As each packet needs only one request, this is not a performance bottleneck, and has the additional benefit of providing intra-switch table consistency trivially. The flow table controller maintains two tables – an exact match table implemented on Block RAMs, and a wildcard match table implemented as a CAM. Each flow table entry consists of three components: a compressed representation of packet header information that serves as a tag for matching against requests, a list of actions determining the output ports and any modifications that need to be made to the matching packet, and flow-specific statistics, *e.g.,* the number of packets in the flow, the number of bytes sent, and the time when the last matching packet was received. The data layout has a one-to-one correspondence with the C-struct in the OpenFlow controller software. The flow table controller pipeline issues a request to both the exact match and the wildcard match tables in parallel, prioritizing the response from the exact match table. If a match is found, it forwards the action list obtained from the matching flow table entry to the action processor module of the corresponding port. If, however, a match is not found, it instructs the action processor to either drop the packet or send the packet to the OpenFlow controller.

Action Processor: The action processor buffers the unmodified packet until it has received the action list from the flow table controller. It updates the destination ports and the packet header, as required by the action list. It can modify the fields of the data link, the network and the transport layers. It also updates the checksum for the network and the transport layers.
Arbiter: The arbiter is an $(N + 1) \times (N + 1)$ crossbar. For every cycle, it selects one flow based on a configurable scheduling policy, and forwards the selected flow to the output queues of all the associated destination ports. This selection is maintained until the entire packet is transmitted.

Switch Controller Interface: The switch controller interface module provides an address-mapped interface for the OpenFlow controller to the flow tables and the statistics. In addition to the necessary logic for marshaling the accesses over the controller-switch communication link, it has interlock logic to guarantee that flow table updates are applied atomically. This allows us to reason about the functionality of the switch at the granularity of packet transfers.

3.2.7 CAMD: Cloud Analysis and Misuse Detection

OpenFlow is an open standard that has gained tremendous interest in the last few years within the network community. It is an embodiment of the software-defined networking paradigm, in which higher-level flow routing decisions are derived from a control layer that, unlike classic network switch implementations, is separated from the data-handling layer. The central attraction to this paradigm is that by decoupling the control logic from the closed and proprietary implementations of traditional network switch infrastructure, researchers can more easily design and distribute innovative flow handling and network control algorithms. Indeed, we also believe that OpenFlow can, in time, prove to be one of the more impactful technologies to drive a variety of innovations in network security. OpenFlow could offer a dramatic simplification to the way we design and integrate complex network security applications into large networks. However, to date there remains a stark paucity of compelling OpenFlow security applications. Here, we introduce *FRESCO*, an OpenFlow security application development framework designed to facilitate the rapid design, and modular composition of OF-enabled detection and mitigation modules. *FRESCO*, which is itself an OpenFlow application, offers a Click-inspired [30] programming framework that enables security researchers to implement, share, and compose together, many different security detection and mitigation modules. We demonstrate the utility of *FRESCO* through the implementation of several well-known security defenses as OpenFlow security services, and use them to examine various performance and efficiency aspects of our proposed framework.

OpenFlow (OF) networks distinguish themselves from legacy network infrastructures by dramatically rethinking the relationship between the data and control planes of the network device. OpenFlow embraces the paradigm of highly programmable switch infrastructures [31], enabling software to compute an optimal flow routing decision on demand. For modern networks, which must increasingly deal with host virtualization and dynamic application migration, OpenFlow may offer the agility needed to handle dynamic network orchestration beyond that which traditional networks can achieve. For an OpenFlow switch, the data plane is made programmable, where flows are dynamically specified within a flow table. The flow table contains a set of flow rules, which specify how the data plane should process all active network flows. In short, OpenFlow's flow rules provide the basic instructions that govern how to forward, modify, or drop each packet that traverses the OF-enabled switch. The switch's control plane is simplified to support the OpenFlow protocol, which allows the switch to communicate statistics and new flow requests to an external OpenFlow network controller. In return, it receives flow rules that extend its flow table ruleset. An OF controller is situated above a set of OF-enabled switches, often on lower-cost commodity hardware. It is the coordination point for the network's flow rule production logic, providing necessary flow rule updates to the switch, either in response to new flow requests or to reprogram the switch when conditions change. As a controller may communicate with multiple OF switches simultaneously, it can distribute a set of coordinated flow rules across the switches to direct routing or optimize tunneling in a way that may dramatically improve the efficiency of traffic flows. The controller also provides an API to enable one to develop OpenFlow applications, which implement the logic needed to formulate new flow rules. It is this application layer that is our central focus. An OF controller is situated above a set of OF-enabled switches, often on lower-cost commodity hardware. It is the coordination point for the network's flow rule production logic, providing necessary flow rule updates to the switch, either in response to new flow requests or to reprogram the switch when conditions change. As a controller may communicate with multiple OF switches simultaneously, it can distribute a set of coordinated flow rules across the switches to direct routing or optimize tunneling in a way that may dramatically improve the efficiency of traffic flows. The controller also provides an API to enable one to develop *OpenFlow applications*, which implement the logic needed to formulate new flow rules. It is this application layer that is our central focus.

From a network security perspective, OpenFlow offers researchers with an unprecedented singular point of control over the network flow routing decisions across the data planes of all OF-enabled network components. Using OpenFlow, an *OF security app* can implement much more complex logic than simplifying halting or forwarding a flow. Such applications can incorporate stateful flow rule production logic to implement complex quarantine procedures, or malicious connection migration functions that can redirect malicious network flows in ways not easily perceived by the flow participants.

Flow-based security detection algorithms can also be redesigned as OF security apps, but implemented much more concisely and deployed more efficiently, as we illustrate in examples.

We introduce a new security application development framework called *FRESCO*. *FRESCO* is intended to address several key issues that can accelerate the composition of new OF-enabled security services. *FRESCO* exports a scripting API that enables security practitioners to code security monitoring and threat detection logic as modular libraries. These modular libraries represent the elementary processing units in *FRESCO*, and may be shared and linked together to provide complex network defense applications. *FRESCO* currently includes a library of 16 commonly reusable modules, which we intend to expand over time. Ideally, more sophisticated security modules can be built by connecting basic *FRESCO modules*. Each *FRESCO* module includes five interfaces: (*i*) input, (*ii*) output, (*iii*) event, (*iv*) parameter, and (*v*) action. By simply assigning values to each interface and connecting necessary modules, a *FRESCO* developer can replicate a range of essential security functions, such as firewalls, scan detectors, attack deflectors, or IDS detection logic.

FRESCO modules can also produce flow rules, and thus provide an efficient means to implement security directives to counter threats that may be reported by other *FRESCO* detection modules. Our *FRESCO* modules incorporate several security functions ranging from simple address blocking to complex flow redirection procedures (dynamic quarantine, or reflecting remote scanners into a honeynet, etc.). *FRESCO* also incorporates an API that allows existing DPI-based legacy security tools (e.g., BotHunter [32]) to invoke *FRESCO's* countermeasure modules. Through this API, we can construct an efficient countermeasure application, which monitors security alerts from a range of legacy IDS and anti-malware applications and triggers the appropriate *FRESCO* response module to reprogram the data planes of all switches in the OpenFlow network.
The *FRESCO* framework consists of an application layer (which provides an interpreter and APIs to support composable application development) and a security enforcement kernel (SEK, which enforces the policy actions from developed security applications. Both components are integrated into NOX, an open source openflow controller.

FRESCO's application layer is implemented using NOX python modules, which are extended through FRESCO's APIs to provide two key developer functions: (*i*) a *FRESCO* Development Environment [**DE**], and (*ii*) a Resource Controller [**RC**], which provides FRESCO application developers with OF switch- and controller-agnostic access to network flow events and statistics. Developers use the *FRESCO script* language to instantiate and define the interactions between the NOX python security modules. These scripts invoke *FRESCO*-internal modules, which are instantiated to form a security application that is driven by the input specified via the FRESCO scripts (e.g., TCP session and network state information) and accessed via *FRESCO's* DE database API.

These instantiated modules are executed by *FRESCO* DE as the triggering input events are received. FRESCO modules may also produce new flow rules, such as in response to a perceived security threat, which are then processed by the controller's security enforcement kernel [**SEK**]. The basic operating unit in the *FRESCO* framework is called a *module*. A module is the most important element of *FRESCO*. All security functions running on *FRESCO* are realized through an assemblage of modules.

A module is implemented as an event-driven processing function. A security function can be realized by a single module or may be composed into a directed graph of processing to implement more complex security services. For example, if a user desires to build a naive *port comparator* application whose function is to drop all HTTP packets, this function can be realized by combining two modules. The first module has *input*, *output*, *parameter*, and *event*. The *input* of the first module is the destination port value of a packet, its *parameter* is the integer value 80, an *event* is triggered whenever a new flow arrives, and *output* is the result of comparing the *input* destination port value and parameter value 80. We pass the *output* results of the first module as *input* of the second module and we assign drop and forward *actions* to the second module. In addition, the second module performs its function whenever it is pushed as an *input*. Hence, the *event* of this module is set to be *push*. A module diagram and modules representing this example scenario are shown in Figure 9.

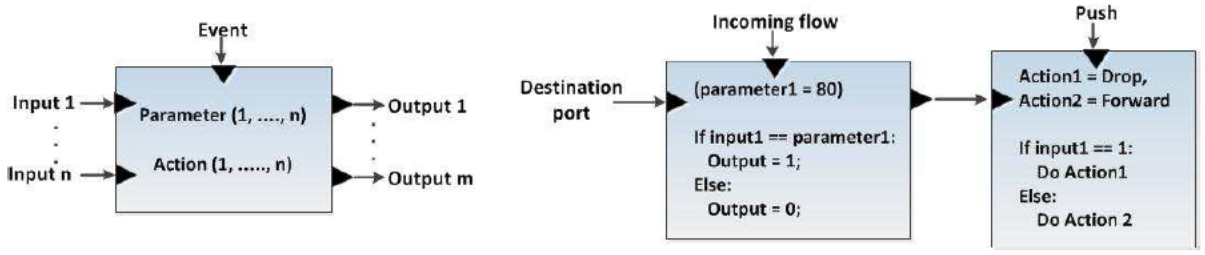


Figure 9: Illustration of FRESCO module design

An *action* is an operation to handle network packets (or flows). The actions provided by *FRESCO* derive from the actions supported by the NOX OpenFlow controller. The OpenFlow standard specifies three required actions, which should be supported by all OpenFlow network switches, and four optional actions, which might be supported by OpenFlow network switches [33]. OpenFlow requires support for three basic actions: (*i*) *drop*, which drops a packet, (*ii*) *output*, which forwards a packet to a defined port, and (*iii*) *group*, which processes a packet through the specified group. As these actions must be supported by all OpenFlow network switches, *FRESCO* also exports them to higher-level applications.

One optional action of interest is the *set action*, which enables the switch to rewrite a matching packet's header fields (e.g., the source IP, destination port) to enable such features as flow path redirection. Because one of the primary goals of *FRESCO* is to simplify development of security functions, *FRESCO* handles possible issues related to the *set action* by breaking the *set action* into three more specific actions: *redirect*, *mirror*, and *quarantine*. Through the *redirect action*, an application can redirect network packets to a host without explicitly maintaining state and dealing with address translation. *FRESCO* offloads session management tasks from applications and automatically changes the source and destination IP address to handle redirects. The *mirror action* copies an incoming packet and forwards it to a mirror port for further analysis. The functionality may be used to send a packet to a feature or other packet analysis systems. The *quarantine action* isolates a flow from the network. Quarantine does not mean dropping a particular flow, but rather, *FRESCO* attaches a tag to each packet to denote a suspicious (or malicious) packet. If a packet has the tag, then this packet can traverse only to allowed hosts (viz., a *FRESCO* script can fishbowl an infected host into an isolated network using packet tags).

The *FRESCO development environment (DE)* provides security researchers with useful information and tools to synthesize security controls. To realize this goal, we design the *FRESCO* DE with two considerations. First, this environment must export an API that allows the developer to detect threats and assert flow constraints while abstracting the NOX implementation and OF protocol complexities. Second, the component must relieve applications from the need to perform redundant data collection and management tasks that are common across network security applications. The *FRESCO* development environment provides four main functions: (*i*) script-to-module translation,(*ii*) database management, (*iii*) event management, and (*iv*) instance execution.

- **Script-to-module translation:** This function automatically translates *FRESCO* scripts to modules, and creates instances from modules, thus abstracting the implementation complexities of producing OF controller extensions. In addition, it is also responsible for validating the registration of modules. Registration is performed via a registration API, which enables an authorized administrator to generate a *FRESCO* application ID and an encryption key pair. The developer embeds the registered application ID into the *FRESCO* script, and then encrypts the script with the supplied private key. The naming convention of *FRESCO* applications incorporates the application ID, which is then used by *FRESCO* to associate the appropriate public key with the application. In addition to registering modules, the module manager also coordinates how modules are connected to each other and delivers input and event values to each module.

- **Database management:** The DB manager collects various kinds of network and switch state information, and provides an interface for an instance to use the information. It provides its own storage mechanism that we call the *FRESCO-DataBase (F-DB)*, which enables one to share state information across modules. For example, if an instance wants to monitor the number of transferred packets by an OpenFlow enabled switch, it can simply request the F-DB for this information. In addition, this database can be used to temporarily store an instance.
- **Event management:** The event manager notifies an instance about the occurrence of predefined events. It checks whether the registered events are triggered, and if so delivers these events to an instance. *FRESCO* supports many different kinds of events, including flow arrivals, denied connections, and session resets. In addition, the event manager exposes an API that enables event reporting from legacy DPI-based security applications, such as Snort [34] or BotHunter [32]. The security community has developed a rich set of network-based threat monitoring services, and the event manager's API enables one to trigger instances that incorporate flow rule response logic.

- **Instance execution:** This function loads the created instances into memory to be run over the *FRESCO* framework. During load time, *FRESCO* decrypts the application using the associated public key, and confirms that the ID embedded in the script corresponds to the appropriate public key. The application then operates with the authority granted to this application ID at registration time.

4.0 RESULTS AND DISCUSSION

4.1 Mirage OS Evaluation

Mirage is a clean-slate implementation of many OS components, and so we evaluate it in stages against more conventional deployments. First, we examine micro-benchmarks to establish baseline performance of key components; and then more realistic appliances: a DNS server, showing performance of our safe network stack; an OpenFlow controller appliance and an integrated web server and database, combining storage and networking. Finally, we examine the differences in active lines of code and binaries in these appliances, and the impact of dead-code elimination.

Microbenchmarks
The purpose of these microbenchmarks is to demonstrate the potential benefits of libOS specialization by examining performance in simple, controlled scenarios; more realistic application benchmarks are provided in subsequent sections. Thus, microbenchmark evaluations are composed of identical OCaml code executing in different hosting environments, labeled as follows: Linux-native, a Linux kernel running directly on the bare metal with an ELF binary version of the application; linux-pv, a Linux kernel running as a paravirtualized Xen domU with an ELF binary version of the application; xen-direct, the application built as a sealed appliance to run directly over Xen, using the Mirage network stack.

Boot Time
Among the benefits of MirageOS is the comparative compactness of the resulting VMs, which significantly reduces domain boot time. Figure 10 compares boot times for a linux-pv Debian Linux VM running only the Apache 2 service, a minimal Linux kernel, and a Mirage unikernel VM. The Debian VM is built using debootstrap and includes only the required runtime packages. The Linux kernel measures the "time to userspace" via a custom-written initrd that calls the ifconfig ioctls directly to bring up a network interface before explicitly constructing and transmitting the single UDP packet required. Time is measured from startup to the point where boot is complete, signaled by the VM sending a special UDP packet to the control domain. For the unikernel and minimal Linux VMs, this is sent as soon as the network interface is ready. For the full Debian VM, it is sent as soon as the Apache process has started. As the memory size increases, the proportion of Mirage boot time due to building the domain also increases, to approximately 60% for memory size 3072 MiB. Mirage matches the minimal Linux kernel, booting in slightly under half the time of the Debian Linux.

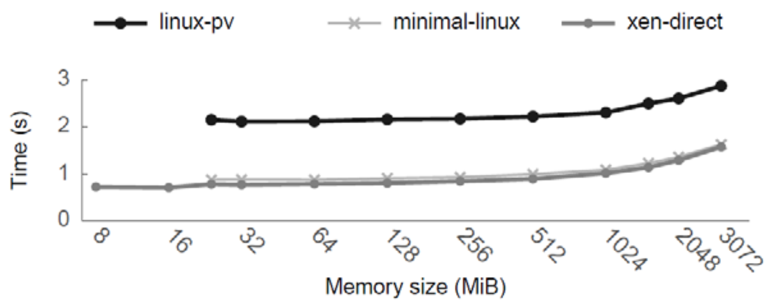
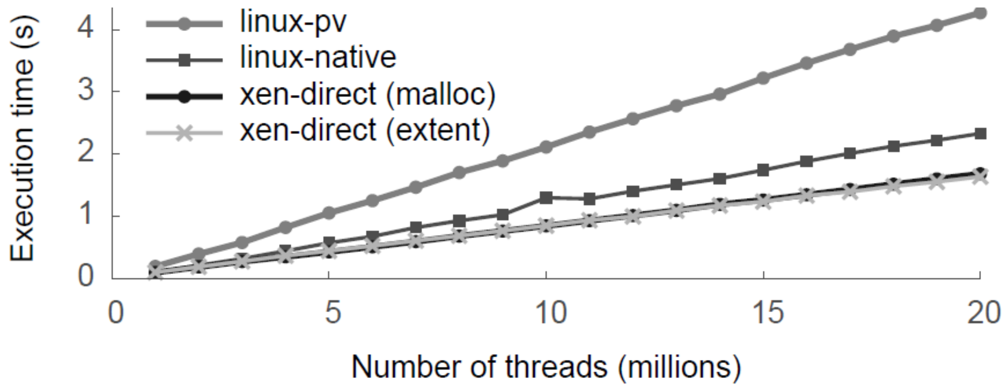


Figure 10: Domain Boot Time Comparison

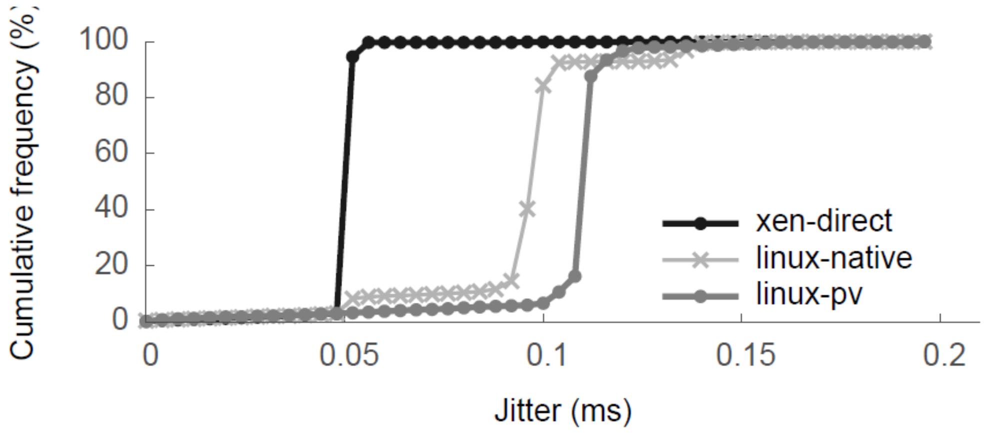
This test is bounded by two factors: the Xen control stack imposes a fixed cost (which affects both Mirage and Linux), and in practice the standard Linux distribution boot time increases if more packages are added and shell scripts become serialized. In contrast, Mirage is jumping directly into a fully functioning high-level language runtime in slightly less time than just the Linux kernel takes to boot, with device drivers synchronously attached.

CPU
We compared performance of an n-body simulation and a page-table stress test via buffer allocation. Detailed results are elided for space but, as expected, performance for the CPU-bound code is unaffected either by type-safety or by lack of a userspace/kernel boundary, as everything runs natively with no emulation. The buffer allocation test stresses page-table manipulation and the Mirage xen-direct implementation slightly outperforms linux-pv due to our lack of a kernel/userspace divide.

Threading
Figure 11a benchmarks thread construction time, showing the time to construct millions of threads in parallel where each thread sleeps for between 0.5 and 1.5 seconds and then terminates. The linux-pv target, which most closely mirrors a conventional cloud application, is slowest with the same binary running on native Linux coming in next. The two xen- targets perform notably better due to the test being limited by the GC speed- thread construction occurs on the heap so creation of millions of threads triggers regular compaction and scanning. The xen- runtime is faster due to the specialized address space layout described earlier. There is little extra benefit to using superpages (xen-extent cf. xen-malloc), as the heap grows once to its maximum size and never subsequently shrinks.We also evaluated the precision of thread timers. A thread records the domain wall clock time, sleeps for 1 to 4 seconds and records the difference between the wall clock time and its expected wakeup time. Figure 11b plots the CDF of the jitter, and shows that the unikernel target provides both lower and more predictable latency when waking up millions of parallel threads. This is due simply to the syscall overhead in Linux, elided by Mirage as there is no userspace/kernel boundary.



(a) Creation times.



(b) Jitter for 10^6 parallel threads sleeping and waking after a fixed period.

Figure 11: Mirage Thread Performance

Networking
As a simple latency test against the Linux stack, we flooded 106 pings from the standard Linux ping client running in its own VM to two targets: a standard Linux VM, and a Mirage application with the Ethernet, ARP, IPv4 and ICMP libraries compiled in. As expected, Mirage performed slightly worse (from 4–10% increase in latency) than Linux, since it implements ICMP in-kernel so there is no userspace/kernel transition to be avoided, while the Mirage stack has the slight overhead of type-safety. Both stacks survived a 72-hour flood ping regression test with no memory leaks. See Table 1.

Table 1: Ping Latency

Platform	*Ping latency (min/avg/max)*					
	56 B packets			1560 B packets		
linux-pv	0.077	0.109	0.264	0.082	0.113	0.272
xen-direct	0.084	0.122	0.299	0.090	0.129	0.284

Figure 12 compares the performance of Mirage's TCP stack against the Linux VM stack. Mirage slightly outperforms Linux when receiving, but does notably worse when transmitting bulk data.

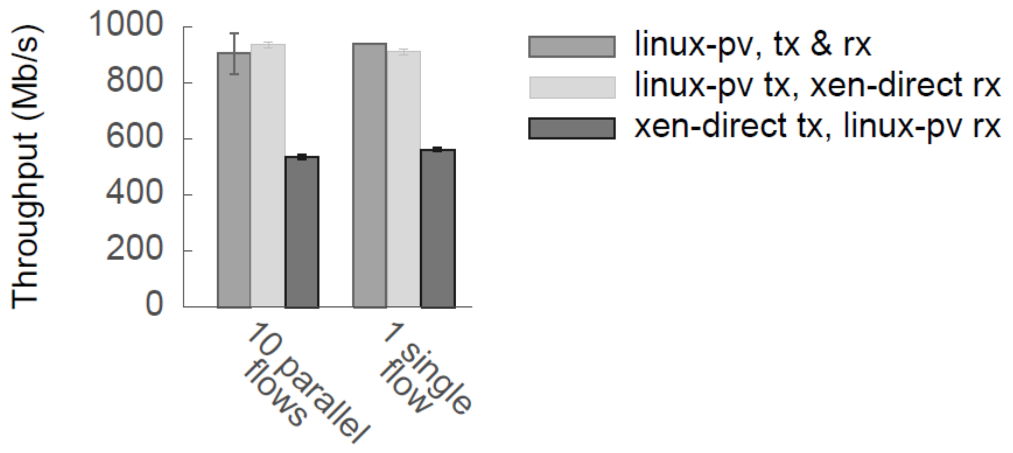


Figure 12: TCP transmit (tx) and receive (rx) throughput over a physical 1 Gb/s Ethernet link

Storage
Figure 13 shows a simple random read throughput test using fio of a fast PCI-express SSD storage device, comparing a Mirage xen-direct appliance against Linux using buffered and direct I/O. Again, as expected, the Linux direct I/O and Mirage lines are effectively the same: both use direct I/O and so impose very little overhead on the raw hardware performance. However, it is notable how big an impact use of the Linux buffer cache has- it causes performance to max out at around 300 MB/sec in contrast to the 1.6 GB/sec achievable if the buffer cache is avoided.

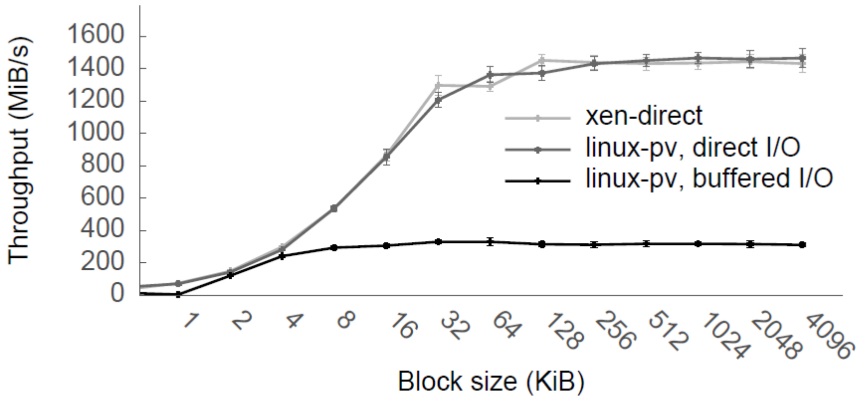


Figure 13: Random block read throughput, t/- 1 std.dev.

To benchmark our OpenFlow implementation we use the OFlops platform [35]. For the controller benchmark we use cbench to emulate 16 switches concurrently connected to the controller, each serving 100 distinct MAC addresses. Experiments run on a 16-core AMD server with 40 GB RAM, with each controller configured to use a single thread. The benchmark measures the throughput in requests processed per second of the controller in response to a stream of packet-in messages produced by each emulated switch under two scenarios, batch and single. Batch is where each switch maintains a full 64 kB buffer of outgoing packet-in messages and single is where only one packet-in message is in flight from each switch. The first measures the absolute throughput when servicing requests, and the second measures throughput of the controller when serving connected switches fairly. Figure 14 compares the xen-direct Mirage controller against two existing OpenFlow controllers: Maestro [36], an optimized Java-based controller; and the optimized destiny-fast branch of NOX [37], one of the earliest and most mature publicly available OpenFlow controllers. Unsurprisingly, the highly optimized NOX fast branch has the highest performance in both experiments, although it does exhibit extreme short-term unfairness in the batch test. Maestro is fairer but suffers significantly reduced performance, particularly on the "single" test, presumably due to JVM overheads. Performance the Mirage appliance falls between NOX fast and Maestro, showing that Mirage manages to achieve most of the performance benefit of optimized C while retaining the high-level language features such as type-safety.

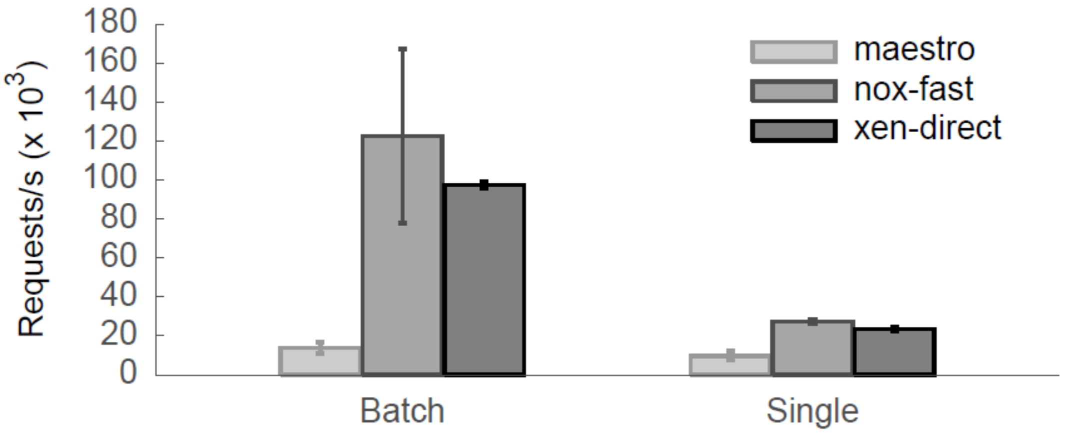


Figure 14: OpenFlow performance comparison

Code and Binary Size
Direct comparison of lines-of-code (LoC) is rarely meaningful due to widespread use of conditional compilation, and complex build systems. We attempt to remove such effects by configuring according to reasonable defaults, and then pre-processing to remove unused macros, comments and whitespace. In addition, to attempt a fair comparison against the 7 million LoC left in the Linux tree after preprocessing, we ignore kernel code associated with components for which we have no analogue, e.g., the many supported architectures, network protocols, and file systems. We are concerned with network-facing guest VMs that share the underlying hypervisor, and so do not include LoC for Xen and its management Table 2: Binary sizes of Xen unikernels, before and after dead-code elimination with configuration and data compiled directly into the

kernel domains, which can be separately disaggregated [4, 38]. Figure 15 shows LoC for several popular server components, taken using the cloc utility. Even after removing irrelevant code, a Linux appliance involves at least 4–5 times more LoC than a Mirage distribution. Note that while the Mirage libraries are not as feature-rich as the industry-standard C applications, their library structure ensures that unused dependencies can be shed at compile time even as features continue to be added (e.g., if no file system is used, then the entire set of block drivers is automatically skipped, in contrast to a Linux distribution where the dependency analysis across the kernel and userspace is non-trivial).

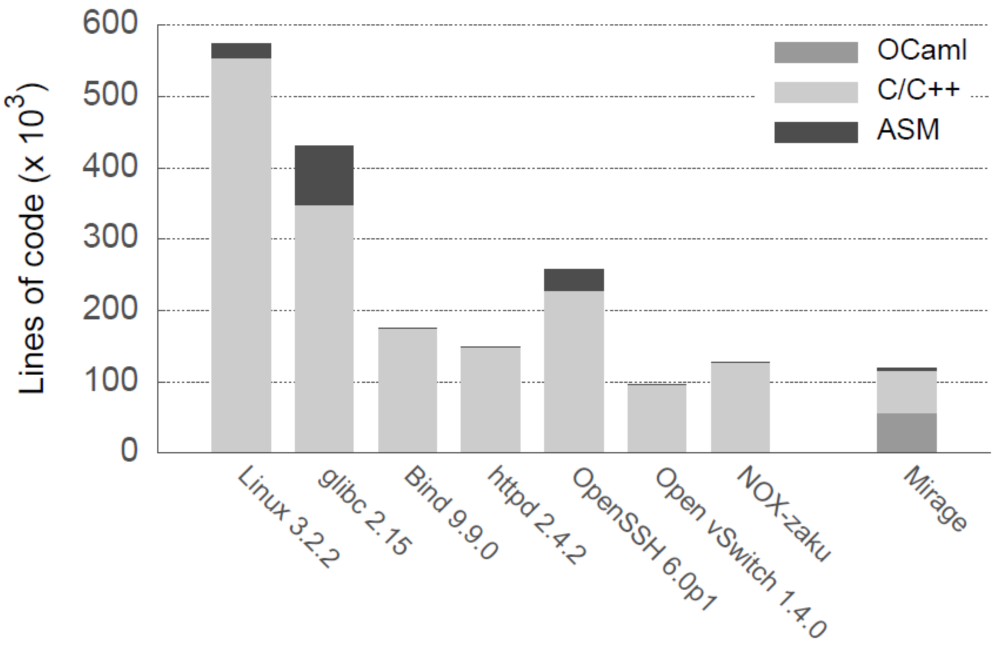


Figure 15: Key cloud components vs. Mirage unikernel codebase

The compiled binary size illustrates this more effectively, and Table 2 lists the earlier appliances. The first column shows the default OCaml dead-code elimination that drops unused modules, and the second is a more extensive custom tool that performs dataflow analysis to drop unused functions within a module if not otherwise referenced (this is safe due to the lack of dynamic linking). Either way, all Mirage kernels are significantly more compact than even a cut-down embedded Linux distribution, and require no special work on the part of the programmer beyond using the Mirage APIs to build their application.

Table 2: Binary sizes of Xen unikernels

Appliance	*Standard build (MB)*	*Dead-code elim (MB)*
DNS	0.449	0.184
Web Server	0.673	0.172
OpenFlow learning switch	0.393	0.164
OpenFlow controller	0.392	0.168

4.2 DIOS Evaluation

In order to maintain portability to different host kernels, the DIOS module never invokes any host kernel functionality directly. Instead, the DIOS Adaptation Layer (DAL) indirects OS-independent requests for kernel functionality (e.g., starting a new process, installing new mappings in the page tables) to the kernel-specific invocations. DAL provides access to data structures and functionality added via the patch. By writing a new DAL for a different host operating system, DIOS can be ported to new platforms. In fact, it should even be possible to run the core of DIOS as a user-space server outside the host OS kernel by implementing a DAL that supports the necessary in-kernel operations.
The DAL currently includes the following functionality that must be supported by host OS kernels:

- Process management: creation and execution of user-space processes, access to and management of DIOS-specific per-process information (usually held in the PCB), retrieval of process information.

- Memory management: allocation of paged and unpaged kernel memory, mapping of kernel memory into user-space virtual address spaces.

- Network access: integration with the kernel network stack, sending and receiving of UDP datagrams (unicast and broadcast).

- Block and character I/O: writing characters to the console.

- Data structures: linked list, hash table, FIFO queue.

- Locking and concurrency: spin locks, reader-writer semaphores.

Since the DAL is largely an adaptation layer, it is fairly compact. The implementation for the Linux kernel consists of about 2,500 lines of C code. Not all OS kernels support loadable modules, however, and different operating system kernels have somewhat different policies as to which symbols kernel modules may access. If loadable kernel modules are not supported, both the DAL and the DIOS core module must be deployed as part of the compile-time kernel patch. If loadable modules are supported, but necessary symbols are not exported to modules, the kernel patch may need to export them. In practice, however, we have not found this to be a problem in the Linux prototype – all symbols required by the DAL come in already exported variants.

As DIOS is ongoing research. We expect to have a comprehensive evaluation of DIOS in the next six months, including a performance evaluation and an evaluation of the security guarantees offered by the DIOS capability model.

4.3 FABLE Evaluation

To compare the performance of different transport protocols in a datacenter environment, we deploy 41 Linux containers on a 48-core Opteron 6168. Connectivity between the containers is enabled through a 1 Gbps virtual switch. We benchmark the performance of scatter-gather applications, which implement tree-based, divide-and-conquer algorithms widely deployed in datacenters [43]. These applications implement a many-to-one communication pattern, which can lead to problems such as incast-induced congestion collapse [44]. We compare TCP Reno, TCP Cubic [45], tcpcrypt [46], DCTCP [47] and MPTCP [48]. These protocols represent a diverse set with fundamentally different congestion control and path usage. For the same application, 3 different flow sizes are used: Large (1 GB), medium (20 MB) and small (10 KB). We use a mix of large flows and bursts of medium and small ones to represent reported datacenter traffic [49]. For all experiments, one container acts as a data sink to receive flows from the rest of the 40 containers.

Figure 16 (a) shows the flow completion time as a function of the flow size, normalized to TCP Reno. The performance of Reno and Cubic is comparable with the latter more amenable to long flows. DCTCP outperforms delay-based congestion control for independent flows irrespective of size. But in case of mixed traffic, it enables bursty short transfers to experience low delay at the expense of longer flows. As a result, the overall flow transfer time undergoes degradation. Multipath TCP on the other hand, improves performance for medium and long flows, but performs poorly for short and bursty traffic. We attribute this to MPTCP's goal of maximizing resource utilization. As a result, short and bursty flows receive a tiny portion of the available network.

We next benchmark the performance of typical datacenter query traffic in the presence of background traffic [47]. To enable this, each container starts a long flow to the sink node, followed by a burst of short flows. The results are presented in 16 (b) DCTCP improves performance for both the short and long flows. In contrast, MPTCP enables the long flows to complete faster than single path Reno and Cubic but short flows experience degraded performance.

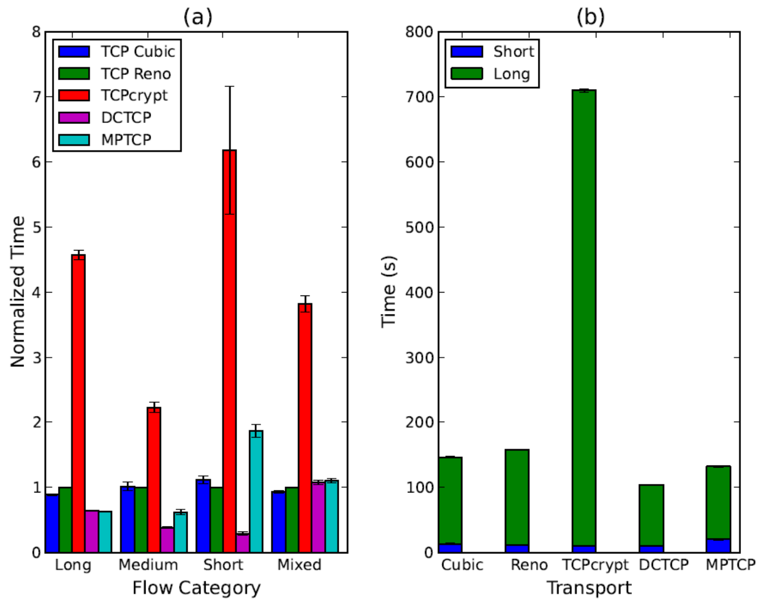


Figure 16: Flow Completion: a comparison of different transports. (a) Flow completion normalized to TCP Reno, (b) flow completion breakdown.

Overall, the results reinforce our position that even in a controlled datacenter environment; the performance of transport protocols is non-uniform.

4.4 RDSF Evaluation

There are close links between RDSF and CHIMERA in that CHIMERA provides the computing infrastructure for RDSF, and RDSF provides the interconnect for CHIMERA. We also plan for a more memory-interconnect style communication for chip-to-chip and board-to-board links. It remains an open challenge for future work to merge Internet-style packet communication with the needs of a cache coherent memory interconnect as this is an ongoing research area of interest. Time did not allow for in-depth evaluations for RDSF.

4.5 CHIMERA Evaluation

Scalability involves some of the test results we would like to acquire. It will determine to what extent we can span the system before the penalties due to link bandwidth, link latency, routing algorithm, boot procedure, and have a negative effect. The other question is, *how sophisticated will the abstraction layer need to be* and *how much FPGA space will this logic consume*? *At what point will routing become a major problem*? When these questions are answered we will be able to say whether this system could be used in a datacenter. If it is possible to scale the system up to the capacity of a typical server rack, we could remove the rack switch completely. If the system can scale beyond a rack we could eliminate other networking elements as well.

Let us assume that the system could scale up to a datacenter rack. A typical rack is 44U in size. Hence, 44 1U servers could be placed in this rack. State-of-the-art server processors consist of 10-12 processor cores, often with two of these processors per server blade. Usually, such processors cannot be fitted into a 1U server blade due to cooling concerns. However, let us assume that we can. This server rack could contain about 1000 processor cores. The rack switch does occupy some space in the rack, but we will ignore that for the purposes of this discussion. To support 44 server blades, at least a 100-port switch is necessary with dual connections to each blade. The latency to the switch will be at best a few hundred clock cycles.

Based on the current implementations of the CHERI and CHERI-2 processors on the DE4 FPGA, a CHERI processor utilizes around 40% of all the LUTs available on the chip. The CHERI-2 utilizes slightly less. Given these conditions, we could potentially have 32 cores per single Bluehive module - a module consists of 16 FPGAs. The complete Bluehive system, four modules, would then run 128 cores. This does not quite match up to the 1000-core rack described earlier. The estimated CHERI/CHERI-2 multi-core system running on the Bluehive would have several factors higher inter-board communication than most commercial systems.

Correctness of distributed cache-coherent memory subsystems presented a particular challenge for this part of the project. Innovation prevailed with exploitation of our BlueCheck test framework married with a memory traffic validation tool, Axe. Axe tests whether coherent memory system meets the criteria for a particular memory consistency model. Such models are challenging to specify a check, so Axe models are written in high-level Haskell and exploit SRI's

Yices constraint solver to efficiently check the results of test-sequences consisting of load, store, load-linked, store-conditional, and memory barrier operations issued by multiple cores. At the time of writing, it supports checking against four consistency models: sequential consistency, total store order, partial store order, and relaxed memory order. This work has been presented in MEMOCODE 2015 [50].

4.6 TPSC Evaluation

The implemented switch operates at 160 MHz and has a 64-bit data path, meeting the 10 Gbps per lane performance requirement. The switch architecture has a pipeline latency of 19 cycles for a packet to travel from ingress to egress. It takes approximately 20% of FPGA LUT/Flip Flop resources and about half of the BRAM resources.

4.7 CAMD Evaluation

For the evaluation, we begin with the basic problem of identifying entities performing flow patterns indicative of malicious network scanning, and compare schemes of implementing network scanning attacks with and without the use of FRESCO.
While network scanning is a well-studied problem in the network security realm, it offers an opportunity to examine the efficiency of entity tracking using FRESCO. Many well-established algorithms for scan detection exist [51, 52, 53]. However, under OpenFlow, the potential for FRESCO to dynamically manipulate the switch's data path in reaction to malicious scans is a natural objective. This scenario also lets us examine how simple modules can be composed to perform data collection, evaluation, and response:

FRESCO Scan Deflector Service. Figure 17 illustrates how FRESCO modules and their connections can be linked together to implement a malicious scan deflector for OpenFlow environments. This scan detection function consists of the three modules described above. First, we have a module for looking up a blacklist. This module checks a blacklist table to learn whether or not an input source IP is listed. If the table contains the source IP, the module notifies its presence to the second module. Based on the input value, the second module performs threshold-based scan detection or it drops a packet. If it does not drop the packet, it notifies the detection result to the third module. In addition, this second module receives a parameter value that will be used to determine the threshold. Finally, the third module performs two actions based on input. If the input is 1, the module redirects a packet. If the input is 0, it forwards a packet. Implementing the three modules required 205 lines of Python code and 24 lines of FRESCO script, as shown in Figure 18.

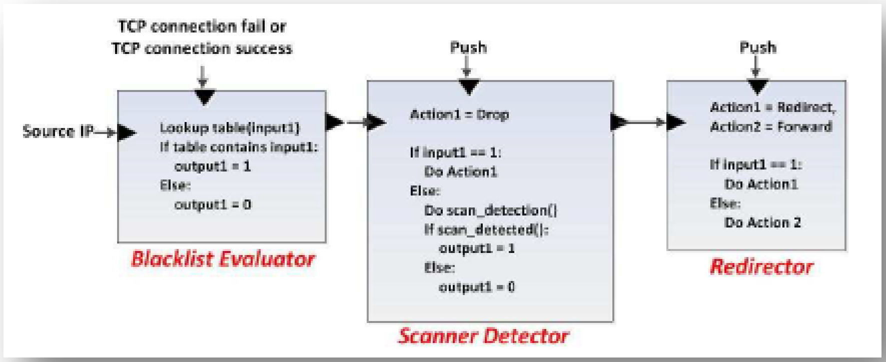
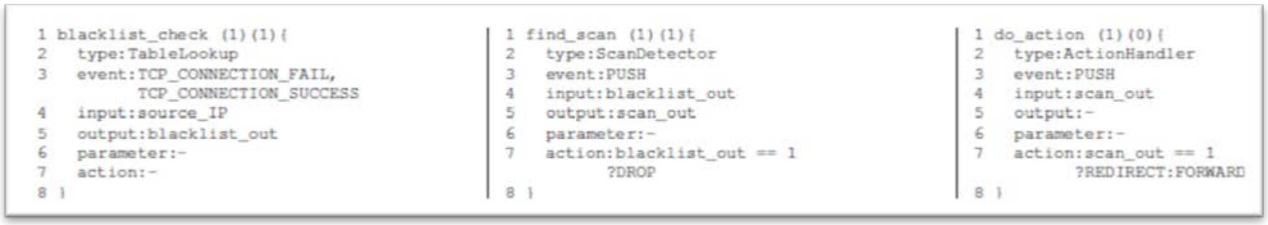


Figure 17: FRESCO composition of a Scan Deflector


```
blacklist_check (1)(1){
  type:TableLookup
  event:TCP_CONNECTION_FAIL,
        TCP_CONNECTION_SUCCESS
  input:source_IP
  output:blacklist_out
  parameter:-
  action:-
}
```

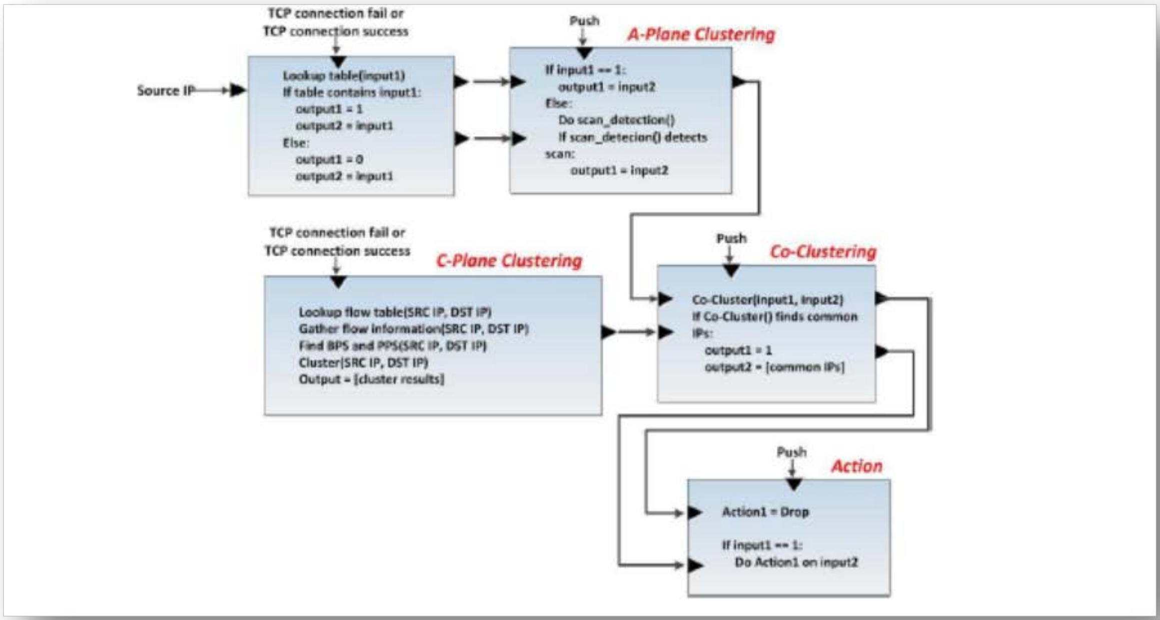
```
find_scan (1)(1){
  type:ScanDetector
  event:PUSH
  input:blacklist_out
  output:scan_out
  parameter:-
  action:blacklist_out == 1
        ?DROP
}
```

```
do_action (1)(0){
  type:ActionHandler
  event:PUSH
  input:scan_out
  output:-
  parameter:-
  action:scan_out == 1
        ?REDIRECT:FORWARD
}
```

Figure 18: FRESCO script for Scan Detector

FRESCO BotMiner Service. To illustrate a more complex flow analysis module using FRESCO, we have implemented a FRESCO version of the BotMiner [54] application. Note that our goal here is not faithful, "bug-compatible" adherence to the full BotMiner protocol [54], but rather to demonstrate feasibility and to capture the essence of its implementation through FRESCO, in a manner that is slightly simplified for readability.

BotMiner detects bots through network-level flow analysis. We have implemented the essentials of its detection functionality using five modules as shown in Figure 19. BotMiner assumes that hosts infected with the same botnet exhibit similar patterns at the network level, and these patterns are different from benign hosts. To find similar patterns between bots, BotMiner clusters botnet activity in two dimensions (C-plane and A-plane). The C-plane clustering approach is used to detect hosts that resemble each other in terms of (packets per second) and bps (bytes per second). The A-plane clustering identifies hosts that produce similar network anomalies. In this implementation, we use the scan detector module to find network anomalies. Finally, if we find two clusters, we perform co-clustering to find common hosts that exist in both dimensions and label them as bots. BotMiner was implemented in 312 lines of python code and 40 lines of FRESCO script, as shown in Figure 20.


TCP connection fail or TCP connection success
Source IP
Push
A-Plane Clustering
TCP connection fail or TCP connection success
C-Plane Clustering
Push
Co-Clustering
Push
Action

```
Lookup table(input1)
If table contains input1:
    output1 = 1
    output2 = input1
Else:
    output1 = 0
    output2 = input1
```

```
If input1 == 1:
    output1 = input2
Else:
    Do scan_detection()
    If scan_detecion() detects
scan:
        output1 = input2
```

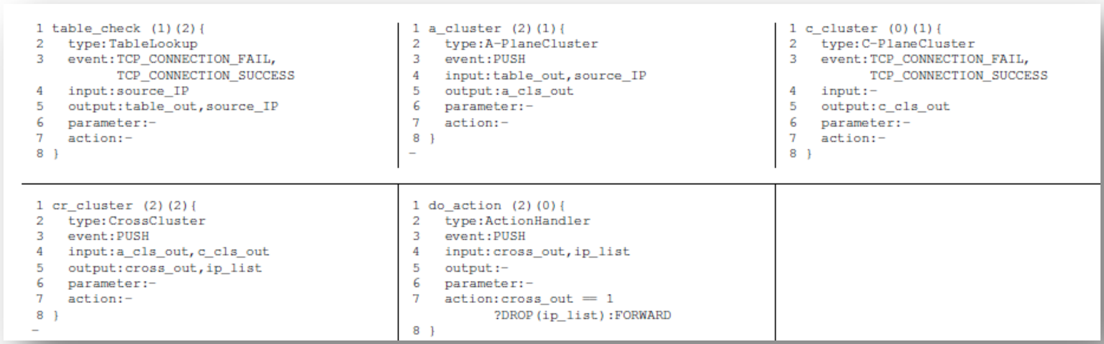
```
Lookup flow table(SRC IP, DST IP)
Gather flow information(SRC IP, DST IP)
Find BPS and PPS(SRC IP, DST IP)
Cluster(SRC IP, DST IP)
Output = [cluster results]
```

```
Co-Cluster(input1, input2)
If Co-Cluster() finds common
IPs:
    output1 = 1
    output2 = [common IPs]
```

```
Action1 = Drop

If input1 == 1:
    Do Action1 on input2
```

Figure 19: FRESCO composition of the BotMiner service



```
table_check (1)(2){
  type:TableLookup
  event:TCP_CONNECTION_FAIL,
        TCP_CONNECTION_SUCCESS
  input:source_IP
  output:table_out,source_IP
  parameter:-
  action:-
}
```

```
a_cluster (2)(1){
  type:A-PlaneCluster
  event:PUSH
  input:table_out,source_IP
  output:a_cls_out
  parameter:-
  action:-
}
-
```

```
c_cluster (0)(1){
  type:C-PlaneCluster
  event:TCP_CONNECTION_FAIL,
        TCP_CONNECTION_SUCCESS
  input:-
  output:c_cls_out
  parameter:-
  action:-
}
```

```
cr_cluster (2)(2){
  type:CrossCluster
  event:PUSH
  input:a_cls_out,c_cls_out
  output:cross_out,ip_list
  parameter:-
  action:-
}
-
```

```
do_action (2)(0){
  type:ActionHandler
  event:PUSH
  input:cross_out,ip_list
  output:-
  parameter:-
  action:cross_out == 1
        ?DROP(ip_list):FORWARD
}
```

Figure 20: FRESCO scripts illustrating composition of the BotMiner service

FRESCO P2P Plotter Service. We have implemented a FRESCO-based P2P malware detection service, similarly implemented to capture the concept of the algorithm, but simplified for the purpose of readability. Motivated by Yen's work [55], we have implemented the P2P malware detection algorithm, referred to as P2P Plotter, using FRESCO. The P2P Plotter asserts that P2P malware has two interesting characteristics, which are quite different from normal P2P client programs. First, P2P malware usually operates at lower volumes of network flow interactions than what is typically observed in benign P2P protocols. Second, P2P malware typically interacts with a peer population that has a lower churn rate (i.e., the connection duration time of P2P plotters is longer than that of normal P2P clients). The algorithm operates by performing co-clustering, to find common hosts that exhibit both characteristics (i.e., low volume and low churn rate). We have implemented this essential functionality of the P2P Plotter algorithm as a 4-module FRESCO script, which is shown in Figure 21. This involved 227 lines of Python code and 32 lines of FRESCO script. The script for the P2P Plotter is illustrated in Figure 22. The reuse of modules (i.e., CrossCluster and ActionHandler, from the BotMiner service implementation is noteworthy, highlighting the reuse potential of FRESCO modules.

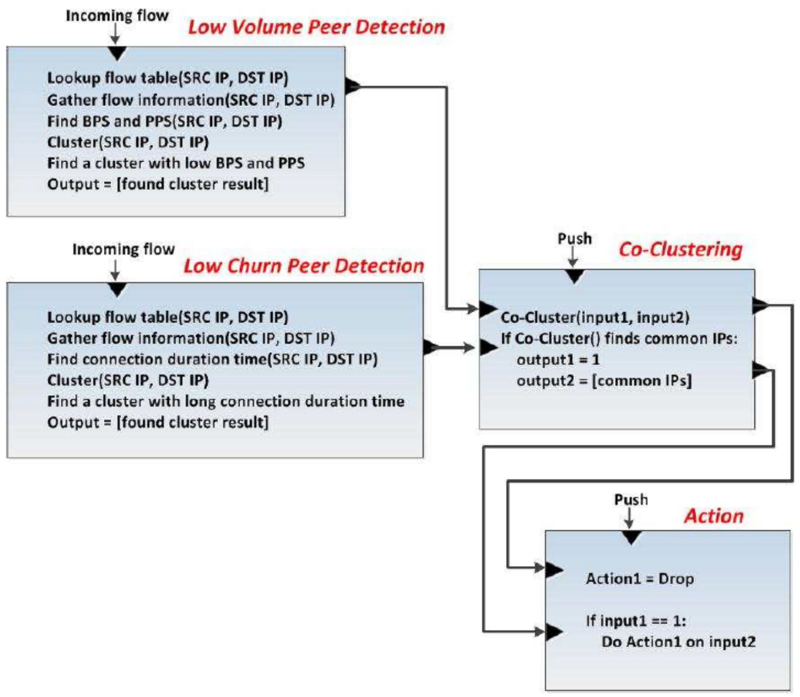
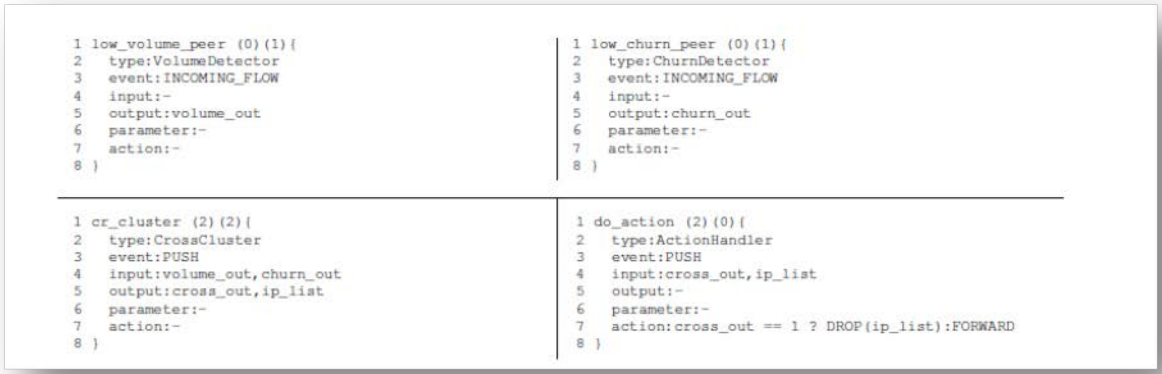

Incoming flow
Low Volume Peer Detection
Lookup flow table(SRC IP, DST IP)
Gather flow information(SRC IP, DST IP)
Find BPS and PPS(SRC IP, DST IP)
Cluster(SRC IP, DST IP)
Find a cluster with low BPS and PPS
Output = [found cluster result]
Incoming flow
Low Churn Peer Detection
Lookup flow table(SRC IP, DST IP)
Gather flow information(SRC IP, DST IP)
Find connection duration time(SRC IP, DST IP)
Cluster(SRC IP, DST IP)
Find a cluster with long connection duration time
Output = [found cluster result]
Push
Co-Clustering
Co-Cluster(input1, input2)
If Co-Cluster() finds common IPs:
output1 = 1
output2 = [common IPs]
Push
Action
Action1 = Drop
If input1 == 1:
Do Action1 on input2

Figure 21: FRESCO composition of the P2P Plotter



```
low_volume_peer (0)(1){
  type:VolumeDetector
  event:INCOMING_FLOW
  input:-
  output:volume_out
  parameter:-
  action:-
}
```

```
low_churn_peer (0)(1){
  type:ChurnDetector
  event:INCOMING_FLOW
  input:-
  output:churn_out
  parameter:-
  action:-
}
```

```
cr_cluster (2)(2){
  type:CrossCluster
  event:PUSH
  input:volume_out,churn_out
  output:cross_out,ip_list
  parameter:-
  action:-
}
```

```
do_action (2)(0){
  type:ActionHandler
  event:PUSH
  input:cross_out,ip_list
  output:-
  parameter:-
  action:cross_out == 1 ? DROP(ip_list):FORWARD
}
```

Figure 22: FRESCO scripts illustrating composition of the P2P Plotter

5.0 CONCLUSIONS

This final technical report provides a comprehensive presentation of the totality of our work over the four years of the (MRC)2 project. We have created a book to document in more detail the entire work effort and application in a 386 pages (MRC)2 Final Project Technical Report Book. There other ideas, projects, and progresses we would have like to have made with (MRC)2, but overall we had a diverse work efforts and with additional time to make various other components well engineered for tech transfer would also be desirable.

5.1 Mirage OS Conclusion

Implementation of the unikernel in Mirage brought to light potential problems with both the underlying approach and the specifics of our implementation path. Perhaps the most obvious is the question of how best to support legacy systems. Our extreme position in this space potentially requires a great deal of re-implementation. Consider for example the SSL protocol and its standard implementation in the OpenSSL [56] library or storage formats such as ext2 that are only completely documented in the code of their standard implementation. A complete re-write of such key components cannot be undertaken lightly! Alternate approaches such as using tools like CIL or CCured [57, 58] to retrofit type safety to existing codebases have been explored by others, but have their own problems. Notably, it is considerable work for them to implement specialization techniques for the particular underlying platform, and it would be hard to integrate the results into a unikernel. At the other end of the spectrum, recent tools such as HipHop [59] take steps toward the unikernel approach, taking PHP code, translating to C++ and then compiling a single binary containing the entire PHP application. One can easily envisage attempting to further specialize that binary into a unikernel, although the benefits of static type safety would not apply with PHP.

Another alternate approach explored in the Flux OSKit [60] is to encapsulate existing code to port it into the new system. In the case of Flux, this was done by targeting the multi-boot bootloader standard and then wrapping device drivers from systems such as Linux to fit within it. Although the greatest benefits of progressive specialization are difficult to obtain in this way, encapsulation is a very promising technique to apply to larger cloud components. For example, ‘big data’ processing systems, such as Map-Reduce, Hadoop, and Dryad [61, 62, 63] are typically structured as a set of intercommunicating processes, and farmed out within a datacenter. Each of these processes could be encapsulated as a single VM and message-passing between VMs implemented via Vchan. This approach is similar to UNIX privilege separation [64], and provides an incremental deployment path, ensuring that existing reliably engineered components can continue to be used and that multiple, new, untested components need not be introduced all at once.

5.2 DIOS Conclusion

In the future, we intend to look into porting DIOS to the CHERI architecture to leverage its native capability support.

5.3 FABLE Conclusion

I/O contention, in both the network and disk interfaces, is a serious problem in many modern environments, especially those making heavy use of virtualization, and significant research effort has gone into handling these issues [65, 66]. The FABLE name service, with its global knowledge of communication patterns, is well-placed to manage these issues. At the most basic level, this could be as simple as selecting a communication channel that is appropriate to the communication environment. For instance, this might mean enabling multi-path TCP on some channels, and deciding which paths to use. These decisions would be constantly re-evaluated by FABLE as the communication environment changes, and, where appropriate, channels would be reconfigured - the APIs presented here allow this to be performed transparently to the overlying application. More interestingly, FABLE can integrate with computation schedulers, at both host and cluster level, to schedule communicating tasks in a way that minimizes contention. Rather than simply reacting to or tolerating contention, FABLE could in many cases prevent it from even occurring. This should allow more efficient use of existing computation resources.

5.4 RDSF Conclusion

As RDSF is ongoing research. We expect to have a comprehensive evaluation of RDSF to include more integration with a technology named SELENA, an experimental Network Simulation Platform, and CHIMERA.

5.5 CHIMERA Conclusion

With the advances in datacenter networks, high-bandwidth physical interconnect standards are emerging that will soon be comparable to the proposed test setup. We intend to produce a performance chart that will demonstrate a scaling ability from 2 to 128 cores. The produced curve should determine the scalability of the system. We can also use multiple threads for every physical core to demonstrate a larger system.

5.6 TPSC Conclusion

Our work in this area thus far has been focused on the TPSC hardware switch and making it a reasonable unit upon which we can do proper analysis of networking. Current switches may drop partially handled packets, and process packets out of order or in a non-serializable fashion. This may be practically acceptable in current network models, but we seek much stronger guarantees in our switch. The eventual TPSC design is intended to provide a hardware switch guaranteeing that each operation is executed such that they are atomic and serializable.

Nirav Dave has written a paper (MEMOCODE 2011) that describes this approach. Essentially, we establish a bi-simulation between this model and the one in which we are interested by leveraging symmetries in the higher-level hardware-description semantic model to make this check relatively straightforward.

This approach is somewhat complicated by the fact that the switch has to interact relatively heavily with the software controller that deals with table updates and other various switch operations that cannot be readily implemented in hardware. Some work has been done on expanding the semantic model to software, effectively letting us unify the reasoning, but this is still work in progress. It seems that we would have a relatively hefty verification problem to get beyond the “hardware is absolutely correct” point to reach the “HW/SW base system is absolutely correct” – at least with respect to its stated requirements. There is certainly much potentially interesting work here, focusing what user abstractions we expose to the switch controller operator, what guarantees and performance possibilities these offer, and how would be use them to implement higher-level protocols and network infrastructure.

5.7 CAMD Conclusion

Despite the success of OpenFlow, developing and deploying complex OF security services remains a significant challenge. We present FRESCO, a new application development framework specifically designed to address this problem. We introduce the FRESCO architecture and its integration with the NOX OpenFlow controller, and present several illustrative security applications written in the FRESCO scripting language. To empower FRESCO applications with the ability to produce enforceable flow constraints that can defend the network as threats are detected, we present the FRESCO security enforcement kernel. Our evaluations demonstrate that FRESCO introduces minimal overhead and that it enables rapid creation of popular security functions with significantly (over 90%) fewer lines of code. We believe that FRESCO offers a powerful new framework for prototyping and delivering innovative security applications into the rapidly evolving world of software-defined networks. We plan to release all developed code as open source software to the SDN community.

6.0 REFERENCES

1. Woodruff, Jonathan et al., “Revisiting RISC in an age of risk”, *International Symposium on Computer Architecture*, Minneapolis, MN, 2014, pp. 457-468.
2. Barham, Paul, et al., “Xen and the Art of Virtualization”, In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, 2003, pp. 164–177.
3. Scott, David et al., “Using functional programming within an industrial product group: perspectives and perceptions”, *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, 2010, pp. 87–92.
4. Colp, Patrick et al., “Breaking up is hard to do: security and functionality in a commodity hypervisor”, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, Portugal, 2011, pp. 189–202.
5. Gazagnaire, Thomas and Hanquez, Vincent, “OXenstored: An efficient hierarchical and transactional database using functional programming with reference cell comparisons”, *SIGPLAN Notices* **44.9**, 2009, pp. 203–214.
6. Baumann, A. et al., “The multikernel: A new OS architecture for scalable multicore systems”, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, 2009, pp. 29–44.
7. Hunt, Galen et al., “Sealing OS processes to improve dependability and safety”, *SIGOPS Operating Systems Review* **41.3**, 2007, pp. 341–354.
8. Ian M. Leslie, Ian M et al., “The design and implementation of an operating system to support distributed multimedia applications”, *IEEE Journal on Selected Areas in Communications* **14.7**, 1996, pp. 1280–1297.
9. David Cheriton, David, “The V distributed system”, *Communications of the ACM* **31.3**, 1988, pp. 314–333.
10. Mullender, Sape J. et al., “Amoeba: a distributed operating system for the 1990s”. *Computer* **23.5**, 1990, pp. 44–53.
11. Ousterhout, John K et al., “The Sprite network operating system”, *Computer* **21.2**, 1988, pp. 23 –36.
12. Pike, Rob, et al., *Plan 9 from Bell Labs*, AT&T Bell Laboratories, Murray Hill, NJ, 1995, p. 9.
13. Hunt, Patrick et al., “ZooKeeper: wait-free coordination for Internet-scale systems”, Proceedings *of USENIX ATC*, Boston, MA, 2010, pp. 1-13.
14. Ganger, Gregory R et al., “Fast and flexible application-level networking on Exokernel systems”, *ACM Trans. Computer System* **20**, 2002, pp. 49–83.
15. Mosberger, David and Peterson, Larry L., “Making paths explicit in the Scout operating system”, *Proceedings of OSDI*, Seattle, Washington, 1996, pp. 153-167.
16. Clark Christopher, et al., “Live Migration of Virtual Machines”, *Proceedings of NSDI*, Boston, MA, 2005, pp. 273-285.

17. Kalyvianaki, Evangelia, et al., “Resource Provisioning for Multi-Tier Virtualized Server Applications”, *Computer Measurement Group Journal* **126,** 2010, pp. 6–17.
18. Barroso, L.A, and U. H¨olzle, U, **The Datacenter as a Computer: An Introduction to the Design of Warehouse Scale Machines**, 2nd ed, San Rafael, California: Morgan & Claypool, 2009.
19. H. Abu-Libdeh, H et al., “Symbiotic Routing in Future Datacenters”, *Proceedings ACM SIGCOMM*, New Delhi, India, 2010, pp. 51-62.
20. Watts, P.M. et al., “Requirements of low power photonic networks for distributed shared memory computers”, *Optical Fiber Communication Conference and Exposition (OFC/NFOEC) 2011 and the National Fiber Optic Engineers Conference*, Los Angeles, CA, 2011, pp. 1 –3.
21. Agarwal, Saurabh, et al., “Adaptive Incremental Checkpointing for massively parallel systems”, *ICS: Proceedings of the 18th Annual International conference on Supercomputing*, Saint Malo, France, 2004, pp. 277–286.
22. Chen, Dong et al., “The IBM Blue Gene/Q interconnection network and message unit”, *High Performance Computing, Networking, Storage and Analysis (SC)*, Seattle, WA, 2011, article 26.
23. Farrington, Nathan et al., “Helios: A hybrid electrical/optical switch architecture for modular data centers”, ACM *SIGCOMM*, New Delhi, India, 2010, pp. 339–350.
24. Perkins, C.E and Royer, E.M, “Ad-hoc on-demand distance vector routing”, *In: Mobile Computing Systems and Applications Proceedings* WMCSA, New Orleans, LA, 1999, pp. 1-9.
25. M.A. Kinsy, M.A. et al., “Heracles: Fully synthesizable parameterized MIPS-based multicore system”, Field *Programmable Logic and Applications (FPL)*, Crete, Greece, 2011, pp. 356-361.
26. Mattson, Timothy G. et al., “The 48-core SCC processor: The programmer’s view”, *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New Orleans, LA, 2010, pp. 1-11.
27. Passas, Stavros, et al., “Towards 100 gbit/s ethernet: Multicore-based parallel communication protocol design”, *ICS: Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, 2009, pp. 214–224.
28. Zhangi, Ying Ping, et al., “A study of the on-chip interconnection network for the PIBM Cyclops64 Multi-Core Architecture”, *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, Rhodes Island, Greece, 2006, pp. 64.
29. Yabe, Tatsuye, “OpenFlow Implementation on NetFPGA-10G- Design Doc.” https://docs.google.com/document/cKwQls6Ted8VZO8h9MjBtu9WxV2fAY44eOgE/edit, Accessed December 24, 2015.
30. Kohler, E et al., “The Click Modular Router”, *ACM Transactions on Computer Systems,* Vol. **18** Issue 3, 2000, pp.263-297.
31. McKeown, Nick, et al., “OpenFlow: Enabling Innovation in Campus Networks”, *Proceedings of ACM Computer Communications Review*, Vol. **38**, 2008, pp. 69-74.

32. Gu, G et al., “BotHunter: Detecting Malware Infection through IDS-driven Dialog Correlation” *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, 2007, article 12.
33. OpenFlow, “OpenFlow 1.1.0 Specification”, http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, Accessed December 24, 2015.
34. Snort, http://snort.org/, Accessed December 24, 2015.
35. Rotsos, C et al., “OFLOPS: An Open Framework for OpenFlow Switch Evaluation”, *PAM'12 Proceedings of the 13th international conference on Passive and Active Measurement*, Vol. **7192**, 2012, pp. 85-95.
36. Cai, Zheng et al., “Maestro: A System for Scalable OpenFlow Control”, *Rice University Technical Report*. TR10-11, Rice University, Houston, TX, 2010.
37. Gude, N et al., “NOX: Towards an operating system for networks,” *SIGCOMM Computer Communications Review*, **38,** 2008, pp. 105–110.
38. Murray, Derek Gordon et al., “Improving Xen security through disaggregation” *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments VEE ’08*, Seattle, WA, 2008, pp. 151–160.
39. Barham, P et al., “Xen and the art of virtualization”, *ACM SIGOPS Operation System Rev*. **37.5**, 2003, pp. 164-177.
40. Baumann, A et al., “The multikernel: A new OS architecture for scalable multicore systems”, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, Montana, 2009, pp. 29–44.
41. Bell, Robert M. et al., *The BellKor solution to the Netflix prize*, AT&T Bell Labs, Florham Park, NJ, 2008.
42. Ben-Yehuda, M et al., “The Turtles Project: Design and Implementation of Nested Virtualization”, Proceedings *of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vol.* **10**, 2010, pp. 423–436.
43. Vamanan, B et al., “Deadline-Aware Datacenter TCP (D2TCP)”, *Proceedings of SIGCOMM 2012*, Helsinki, Finland, 2012, pp. 115-126.
44. Vasudevan, V et al., “Safe and effective fine-grained TCP retransmissions for datacenter communication”, *ACM SIGCOMM CCR*, Vol. **39-4**, 2009, pp. 303-314.
45. Sangtae Ha et al., “CUBIC: A new TCP-friendly high-speed TCP variant” *SIGOPS Operational*, Rev.**42.5**, 2008, pp. 64-74.
46. A. Bittau et al., “The case for ubiquitous transport-level encryption”, *Proceedings of the 19th USENIX conference on Security USENIX Security’10*, Washington, DC, 2010, pp.26.
47. M. Alizadeh et al., “Data center TCP (DCTCP)”, *ACM SIGCOMM CCR*, Vol. **40**, ACM, 2010, pp. 63-74.
48. Raiciu, C et al., “How hard can it be? Designing and implementing a deployable multipath TCP”, *Proceedings of NSDI’12*, San Jose, CA, 2012, pp. 1-14.

49. T. Benson et al., “Network traffic characteristics of data centers in the wild”, ACM *SIGCOMM IMC’10*, Melbourne, Australia, 2010, pp. 267-280.
50. M. Naylor and S.W. Moore, “A Generic Synthesisable Test Bench”, *Formal Methods and Models for Codesign (MEMOCODE) 2015 Thirteenth ACM/IEEE International Conference*, Austin, TX, 2015, pp. 128-137.
51. J. Jung et al., “On the Adaptive Real-time Detection of Fast Propagating Network Worms” *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA),* Lucerne, Switzerland, 2007, pp. 175-192.
52. Jaeyeon Jung, et al., “Fast Portscan Detection Using Sequential Hypothesis Testing”, *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2004, pp. 211-225.
53. Vyas Sekar et al., “A Multi-Resolution Approach for Worm Detection and Containment”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Philadelphia, PA, 2006, pp. 189-198.
54. Guofei Gu, et al., “BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection”, *Proceedings of the USENIX Security Symposium (Security’08),* San, Jose, 2008, pp. 139-154.
55. T.-F. Yen and M. K. Reiter, “Are Your Hosts Trading or Plotting? Telling P2P File-sharing and Bots Apart”, *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, 2010, pp. 241-252.
56. The OpenSSL Project, OpenSSL, http://openssl.org/, Accessed April 2012.
57. Necula, George C et al., “CIL: Intermediate language and tools for analysis and transformation of C programs”, *Proceedings of the 11th International Conference on Compiler Construction CC ’02*, Grenoble, France,2002, pp. 213–228.
58. Necula, George C et al., “CCured: Type-safe retrofitting of legacy code”, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL ’02*, Portland, Oregon, 2002, pp. 128–139.
59. Facebook, “HipHop for PHP”, https://github.com/facebook/hiphop-php/wiki/, Accessed Feb 2010.
60. Ford, Bryan et al., “The Flux OSKit: A substrate for kernel and language research”, *Proceedings of the 16th ACM Symposium on Operating Systems Principles SOSP ’97*, Saint Malo, France, 1997, pp. 38–51.
61. Apache, “Hadoop”, http://hadoop.apache.org, Accessed April 2012.
62. Dean, Jeffrey and Ghemawat, Sanjay, “MapReduce: Simplified Data Processing on Large Clusters”, *Communications of the ACM*, **51.1,** 2008, pp. 107–113.
63. Isard, Michael et al., “Dryad: distributed data parallel programs from sequential building blocks”, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 EuroSys ’07,* Lisbon, Portugal, 2007, pp. 59–72.

64. Provos, Niels et al., “Preventing privilege escalation”, *Proceedings of the 12th Conference on USENIX Security Symposium SSYM’03*, Washington, DC, 2003, pp. 231–242.
65. Mohammad Alizadeh, et al. “Data Center TCP (DCTCP)”, *Proceedings of SIGCOMM*, New Delhi, India, 2010,pp. 63–74.
66. Raiciu,Costin et al., “Improving datacenter performance and robustness with multipath TCP”, *Proceedings of SIGCOMM*, Toronto, Ontario, Canada, 2011, pp. 266–277.

7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

BSV – Bluespec SystemVerilog, developed by Bluespec Inc. to enable the use of the BSV compiler to transform hardware specifications written in the BSV specification language into a form that can be executed on FPGAs or simulated.

CHERI -- Capability Hardware Enhanced RISC Instructions; this acronym is used with respect to the CHERI hardware Instruction-Set Architecture (ISA) and the CHERI system architecture, among other entities.

CRASH – Clean-slate Resilient Adaptable and Secure Hosts; this is the DARPA program under which the CTSRD project operated.

CTSRD – CRASH-worthy Trustworthy Systems Research and Development

FPGA – Field-Programmable Gate Arrays, which provide the ability to execute an instruction-set architecture as if it were real hardware

ISA – Instruction-Set Architecture

MRC – Mission-oriented Resilient Clouds, a companion DARPA program to CRASH

(MRC)^2 (pronounced MRC-squared) – Modular Research-based Composably trustworthy Mission-oriented Resilient Clouds; this SRI-Cambridge MRC project encompassed some clean-slate approaches to secure software-defined networking (SDN) and trustworthy cloud servers, among many other innovations and developed prototypes.