Sikker: A High-Performance Distributed System Architecture for Secure Service-Oriented Computing

Nicholas McDonald and William J. Dally

Stanford University

{nmcdonal,dally}@stanford.edu

ABSTRACT

In this paper, we present Sikker[1], a high-performance distributed system architecture for secure service-oriented computing. Sikker includes a novel service-oriented application model upon which security and isolation policies are derived and enforced. The workhorse of Sikker is a custom network interface controller, called the Network Management Unit (NMU), that enforces Sikker's security and isolation policies while providing high-performance network access.

Sikker's application model satisfies the complex interactions of modern large-scale distributed applications. Our experimentation results show that even when implemented on very large clusters, the NMU adds a negligible message latency of 41 ns under realistic workloads and 66 ns at the 99^{th} percentile of worst case access patterns. Our analysis shows that the NMU can easily support over 100 Gbps with a single logic engine and that over 500 Gbps is achievable with more aggressive designs.

Sikker's service-oriented security and isolation mechanism removes high overheads imposed by current systems. Sikker allows distributed applications to operate in an environment with fine-grained security and isolation while experiencing supercomputer-like network performance.

1. INTRODUCTION

The number and variety of applications and services running in modern data centers, cloud computing facilities, and supercomputers has driven the need for a secure computing platform with an intricate network isolation and security policy. Traditionally, supercomputers focused on performance at the expense of internal network security while data centers and cloud computing facilities focused on cost efficiency, flexibility, and TCP/IP compatibility all at the expense of performance. In spite of their historical differences, the requirements of these computing domains are beginning to converge. With increased application complexity, data centers and cloud computing facilities require higher network bandwidth and predictably low latency. As supercomputers become more cost sensitive and are simultaneously utilized by many clients, they require a higher level of application isolation and security. The advent of cloud-based supercomputing [3, 18] brings these domains even closer by merging them onto the same network.

Operating under a single administrative domain allows distributed systems to consider the network a trusted entity and rely on its features. Supercomputers use this ideology to achieve ultimate performance, however, they maintain minimal security and isolation mechanisms. In contrast, cloud computing facilities achieve high levels of security and isolation at the expense of much lower performance. In theory, a single administrative domain *could* provide simultaneous performance, security, and isolation as these are not fundamentally in opposition. The unfortunate truth is that modern network technologies have not provided distributed systems that are capable of supercomputer-like network performance while simultaneously providing robust application security and isolation. As a result, system designers and application developers are forced to make trade-offs leaving deficiencies in their system and creating high development and runtime overheads.

In this paper, we present a new distributed system architecture called Sikker, that includes an explicit security and isolation policy. The goal of this system is to provide the highest level of network performance while enforcing the highest level of application security and isolation required by the complex interactions of modern large-scale applications. Sikker formally defines a distributed application as a collection of distributed services with well-defined interaction policies. Sikker utilizes specially architected network interface controllers (NICs), called Network Management Units (NMUs), to enforce application security and isolation policies while providing efficient network access. Unlike common NICs, NMUs operate directly under the control of a system-wide Network Operating System (NOS), and as such, are not vulnerable to compromises of individual host operating systems.

This paper makes the following contributions:

- We present a new distributed system security and isolation model that is built from a service-oriented access control list methodology. This is the first work

[1] Sikker is a danish translation for "safe"

to present a service-oriented network architecture and process-oriented authentication.

- We show how modern large-scale applications fit into this model and how they can be modified to make use of it.
- We present the Network Management Unit, a high performance network interface controller that, under the direction of a Network Operating System, enforces the security and isolation policies of Sikker.
- We provide an evaluation of Sikker and the NMU which shows that it simultaneously provides high performance network access and robust application security and isolation.

The outline of this paper is as follows. In Section 2 we discuss the motivation of Sikker in terms of performance and application structure and propose a new methodology for implementing access control. In Section 3 we describe the Sikker architecture as an abstract system with strict requirements and plentiful features. In Section 4 we present the Network Management Unit as the device that enables Sikker to operate as described with high performance. Section 5 describes our evaluation methodology and Section 6 shows the evaluation results of Sikker and the NMU. Section 7 presents prior related work and in Section 8 we conclude.

2. MOTIVATION

2.1 Attainable Performance

The highest level of network performance available today is found in supercomputing interconnection networks such as Cray Cascade [14] and Gemini [1], IBM Blue Gene/Q [12] and PERCS [5], and Mellanox InfiniBand [24]. These interconnects achieve high bandwidth and predictably low latency while incurring minimal CPU overhead. For example, InfiniBand networks manufactured by Mellanox Technologies achieve round-trip times on the order of 2 µs and bandwidths as high as 100 Gbps [24]. The Cray Cascade system achieves unidirectional latencies as low as 500 ns and provides 93.6 Gbps of global bisection bandwidth per node [14]. In order to achieve our goal of high network performance, we define our metrics for performance relative to the highest performing interconnection networks.

One of the major strategies that supercomputers use to achieve high performance is allowing applications to bypass the operating system and interact with the network interface directly. This is called *OS-bypass*. All major high performance computing fabrics (e.g. Cray, IBM, Mellanox, Myricom, Quadrics) have taken this approach. Along with providing lower and more predictable network latency, OS-bypass provides lower CPU overhead as the kernel is freed of the task of managing network interface sharing. CPU overhead can be further reduced by offloading network transport protocols to the network interface.

OS-bypass has one major ramification, namely, bypassing the kernel (or hypervisor) removes its ability to monitor, modify, rate limit, or block outgoing network traffic in an effort to provide sender-side security and isolation features as is commonly performed in network virtualization software.

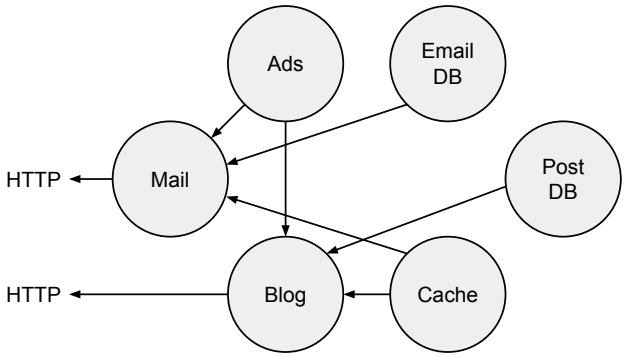

Figure 1: High level service connectivity. Directed edges show direction of functionality offering.

2.2 Service-Oriented Applications

Modern large-scale distributed applications are often comprised of thousands of processes. For reasons of management, separation of development, modularity, and fault tolerance, these processes are grouped by similarity into collections called *services*. A service is a collection of processes developed and executed for the purpose of implementing a subset of an application's functionality. Applications can be comprised of one or more services, often tens or hundreds, and services are often shared between many applications. Figure 1 shows a simplified diagram of six services interacting to fulfill the functionality of two user facing applications, an email system and a blogging system. Each service has a defined application programming interface (API) that it exposes to provide functionality to other services. Even though a modern data center might contain thousands of services, each service generally communicates with a small subset of the total services in order to fulfill its designed functionality. Furthermore, it is common for a service to use only a portion of another service's API.

Figure 2 is a diagram created by Twitter to illustrate the operation of their protocol-agnostic communication system. Similarly, Figure 3 is a diagram created by Netflix illustrating their architecture on Amazon's cloud computing platform. For both of these designs, there exists several services custom written for the application, as well as several services written by third-parties. Both of these diagrams show that when designing an application at a high level, appli-

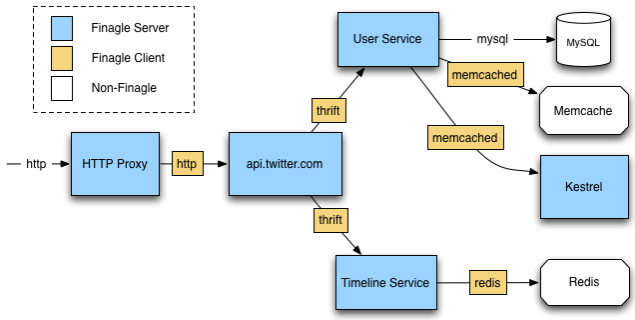

Figure 2: Twitter's Finagle RPC system[42].

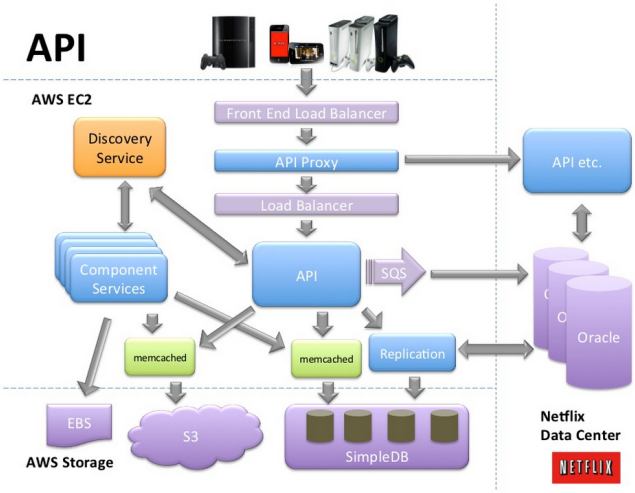

Figure 3: Netflix's architecture on Amazon's AWS [29].

cation developers divide the application's functionality into services with well-defined APIs to achieve modularity.

An inspection of the code of any given service would reveal the implicit interaction privileges it desires with other services. In most cases, the code expressing the desired interactions does not contain IP addresses or TCP port numbers, but instead contains service names, process identifiers, permission domains, and API commands. For example, from the Twitter example in Figure 2 we might see the *Timeline Service* desiring to communicate with the *Redis* service using its process #6 and using API command *Get*.

2.3 Access Control

The implicit permissions, discussed in Section 2.2, declared by each service present the ideal level at which permissions should be enforced as these permissions are derived from the applications themselves and represent the actual intent of the services on the network. The available security and isolation techniques in today's data centers use multiple layers of indirection before permissions are checked and enforced. This creates high operational complexity and overheads and presents many opportunities for misconfiguration. These systems lose information about the original intent of the application, thus, cannot enforce the permission as it was intended. The lack of inherent identity authenticity within the network forces developers to use authentication mechanisms (e.g. cryptographic authentication) that incur high CPU overhead and are unable to properly guard against denial-of-service attacks due to the lack of isolation. In this section, we will describe how current systems work and our proposal for a better solution.

To moderate network access, modern network isolation mechanisms use access control lists (ACLs). In the abstract form, an ACL is a list of entries each containing identifiers corresponding to a communication mechanism and represent a permissions whitelist. For access to be granted, each communication must match on an entry in the ACL. The most common type of ACL entry is derived from TCP/IP network standards. We will further refer to this style of ACL as a network-ACL or NACL. Table 1 shows an example of an NACL entry commonly represented as a *5-tuple*. This entry states that a packet will be accepted by the network if the protocol is TCP and it is being sent from 192.168.1.3 port 123 to 10.0.2.10 port 80. Portions of a NACL can be masked out so that only a portion of the entry must be matched in order for a packet to be accepted by the network.

A comparison between the NACL whitelisting mechanism and the implicit permissions discussed in section 2.2 exposes the deficiencies of using any ACL system based on network-centric identifiers such as protocols, network addresses, or TCP/UDP ports. One important thing to notice is that the source entity is referenced by an IP address and optionally a port. For this system to work as desired, the system must know with absolute confidence that the source entity is the only entity with access to that address/port combination and that it is unable to use any other combination. This is hard to ensure because the notion of an IP address is very fluid. While it is commonly tied to one NIC, modern operating systems allow a single machine to have many NICs, NICs can have more than one IP address, and/or multiple NICs can share one or more IP addresses. There is no definitive way to determine the source entity based solely from a source IP address. Another issue is the use of UDP and TCP ports, which are abstract identifiers shared among all the processes on a given machine. Tying the permissions to ports requires the source and destination to keep numerous open sockets proportional to the number of permission domains required by the application.

ACL whitelisting has the right intent with its approach to security and isolation because of its inherent implementation of the principle of least privilege [36] and its ability to prevent denial-of-service attacks by filtering invalid traffic *before* it enters the network. However, using network-centric ACLs is the source of security and isolation deficiency in modern networks.

In order to design a better system, we propose creating ACL entries based directly from the privileges discussed in section 2.2. Our ACL entries exactly express the communication interactions of services and their APIs. We will further refer to this style of ACL as a service-ACL or SACL. Table 2 shows an example of a pair of SACL entries which reference the source entity by its actual identity, the service. The destination is also referenced by the service along with the process identifier within the service and the permission domain to be accessed. As shown, two entries are needed to communicate with a destination as each communication desires to connect with a destination process and a destination permission domain. This creates a permission orthogonality between processes and domains. In this example, repeated from the Twitter example from Figure 2, the *TimelineService* has been given access to the *Redis* service using process #6 and using the *Get* permission domain. SACLs make reasoning about network permissions much easier and don't tie the

Protocol	**Source**		**Destination**	
	Address	*Port*	*Address*	*Port*
TCP	192.168.1.3	123	10.0.2.10	80

Table 1: A network-centric ACL (NACL) entry.

Source	Destination		
Service	*Service*	*Process*	*Domain*
TimelineService	Redis	6	-
TimelineService	Redis	-	Get

Table 2: Example service-oriented ACL entry (SACL).

permission system to any underlying transport protocol or addressing scheme. It simply enforces permissions in their natural habitat, the application layer.

A tremendous amount of security and isolation benefits are available to the endpoints if the following system-level requirements are upheld for the SACL methodology:

SACL Requirements:.

S.1 The network is a trusted entity and no endpoint has control over it.

S.2 The network is able to derive the identity of a process and it is impossible for a process to falsify its identity.

S.3 The source (sender) identifier is sent with each message to the destination (receiver).

S.4 Messages sent are only received by the specified destination entity.

With these requirements upheld, the system inherently implements source authentication by which all received messages explicitly state the source entity's identification. Destination authentication is also inherent by the same logic. Combined, source and destination authentication remove the need for complex authentication software in the application layer. Furthermore, senders don't need to use name servers to discover physical addressing for desired destinations as they only need to specify the destination by its virtual identity (i.e. service ID, process ID, and domain ID) and the network will deliver the message to the proper physical location.

3. SIKKER

3.1 Application Model

With the insights gained in section 2, we define a new distributed system architecture, called *Sikker*, that formally defines the structure of distributed applications. Sikker is strictly a service-oriented architecture and makes no attempt to justify the boundaries of *applications*. As a service-oriented architecture, Sikker designates the *service* as the fundamental building block of distributed applications.

Each service in Sikker contains a set of *processes* as its execution units that implement a common API. A process can be an OS process, software container, virtual machine, etc. Each process within a service is assigned a numerical ID unique to the service.

The API of each service in Sikker contains a set of permission *domains*, subsequently referred to as *domains*. Each domain represents a portion of the service's functionality with respect to a specific permission. Sikker domains are not used for multiplexing, are not shared, and are only used to specify a destination. Each service has its own domain number space, thus, two services using the same domain ID is acceptable.

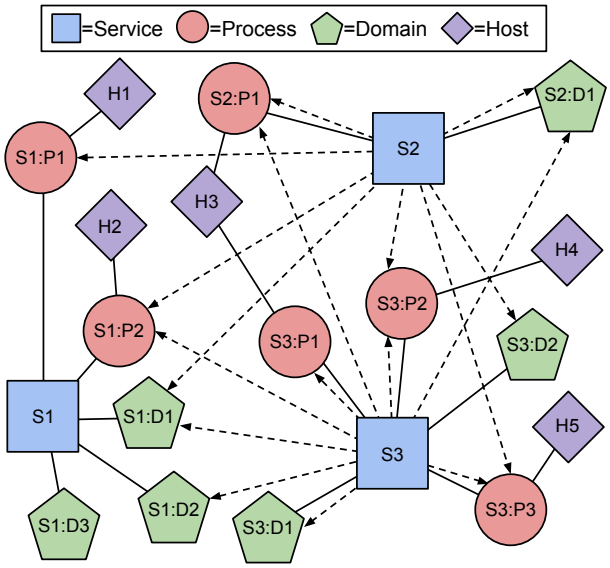

Figure 4: An example service interaction graph. Solid edges represent assignment and dashed edges represent permissions.

To understand the usage of Sikker domains, consider a simple key/value storage service that exposes functionality to perform data retrieval like the "get" command in memcached or Redis. Assuming that the service only allows users to access their own data and not data stored by other clients, the service would need to define a domain for the data retrieval function per client. Thus, for a system with three clients there would be three domains for the "get" functionality, one for each user's data. This example shows how a service might tie its specific API commands to domains. An alternative is to group the API commands into access types (e.g. read and write) which results in fewer total domains. Another alternative is to only create one domain per user. All these are acceptable schemes in Sikker but will yield different granularities on which security and isolation can be enforced.

Figure 4 is an example of service interactions under the Sikker application model. This diagram shows three services, each with a few processes and a few domains. Solid lines connect services to their corresponding processes and domains as well as connect processes to their corresponding hosts. As shown, and widely used in practice, processes from the same service and/or different services may overlap on the same host. Dashed lines show the permissions given to services. These lines originate at a service and end at a process or a domain.

Each process within a service inherits all the permissions of the service to which it belongs. In order a process to be able to transmit a message to a specific destination, the service of the sending process must have permission to access the specified process and domain within the specified des-

tination service. Sikker performs permission checks before messages enter the network and for every message. Because the interaction policies of modern large-scale distributed systems are constantly in flux, Sikker allows processes and domains to be added and removed from services dynamically during runtime. When a new process is created, it inherits all the permissions of the service to which it belongs. Any time the permissions of a given service change, the change is reflected in all processes of the service.

3.2 Authentication

All communication in Sikker is explicitly authenticated at the source and destination. Similar to other networks, processes in Sikker reside at physical locations specified by physical addresses. However, in Sikker, processes are referenced by virtual addresses that specify both the service and the process. When a process desires to send a message on the network, it does not specify its own identity as the source. Instead, Sikker derives its identity, consisting of both service and process, and attaches it to the message.

When specifying a destination for a message, the source process specifies the destination by three things: a service, a process within the service, and a domain within the service. Combined, the source and destination specifications are attached to every message transmitted on the network. Sikker guarantees that the message will only be delivered to the specified destination. Receiving processes are able to inspect the source specification in the message to explicitly know the source's identity.

Under the Sikker security model, processes need not be concerned about physical addressing in the network. Processes only use service-oriented virtual network addresses when referencing each other. Sikker performs the virtual-to-physical translations needed for transmission on the network. There is no need for name servers in Sikker.

3.3 One-Time-Permissions

The use of request-response protocols are ubiquitous in service-oriented applications. In this environment, many services only become active when they receive requests from other services. This master/slave interaction is achieved via request-response protocols. Cloud computing providers often provide services like this with many features to increase the productivity of their tenants. These services (e.g. Amazon S3[4], Google BigTable[11], Microsoft Azure Search[26]) can be very large and provide functionality to many thousands of clients.

To increase scalability and to fit better with large-scale request-response driven multi-tenant systems, Sikker contains a mechanism for one-time-permissions (OTPs). An OTP is a permission generated by one process and given to another process to be used only once. An OTP specifies a service, process, and domain as a destination and can only be created using the permissions that the creating process already has. When a process receives an OTP from another process, it is stored by Sikker in a temporary storage area until it gets used by the process, at which time Sikker automatically deletes the permission. Because an OTP fully specifies the destination, the process using it specifies the OTP by its unique ID instead of specifying the destination as a service, process, and domain. Only the process that received the OTP can use it. OTPs cannot be shared across the processes in a service.

For an example of using OTPs, consider Service 1 in Figure 4 which has no permissions assigned to it, thus, cannot send messages on the network. Assume this service is a simple in-memory cache service. Its API specifies that users of the service must give it an OTP with each request. Now assume that Service 2 Process 1 (S2,P1) wishes to send a request to Service 1 Process 2 Domain 1 (S1,P2,D1). When it formulates its request, it generates an OTP that specifies itself (S2,P1) with Domain 1 as the recipient (S2,P1,D1). (S1,P2) will receive the OTP with the request and when the response is ready to be sent, it simply uses the OTP to send it. After the response is sent, Sikker deletes the OTP.

Another interesting example of using OTPs is allowing one service to act on behalf of another service. Given the same example as before, assume that (S2,P1) wants the response to be sent to (S3,P3,D2) instead of itself. Because it has the proper permissions, it is able to create the OTP with this recipient. The effect is that (S2,P1) sends the request to (S1,P2,D1), then (S1,P2) sends the response to (S3,P3,D2).

3.4 Network Operating System

Sikker requires the existence of a network operating system (NOS) to act as a trusted system-wide governor. The NOS creates the services running on the network, establishes their permissions, and distributes the proper permissions to the proper entities in the system. The NOS is externally reachable such that users are able to start new services on the system and control existing services that they own. While interacting with the NOS, the user is able to specify the structure of a new service in terms of processes and domains. Furthermore, the user is able to create fine-grained permission sets (a set of processes and a set of domains) which other services will be able to use. During runtime, services are able to contact the NOS for changes to their own structure and for permission changes. The specific placement, implementation, fault tolerability, and user interface of such a NOS is beyond the scope of this work.

4. NETWORK MANAGEMENT UNIT

4.1 Architecture

In this section, we present the Network Management Unit (NMU), a new NIC architecture that is the workhorse of Sikker. The NMU provides each process with high-performance network access while implementing the Sikker security and isolation model, described in Section 3. The NMU can be viewed as an extension to the standard NIC architecture with the following requirements:

NMU Requirements:.

N.1 A method for efficient interaction between local processes and the network.

N.2 A method of deriving the identity of local processes using the network.

```
IndexMap: (LocalService, LocalProcess) → LocalIndex
InfoMap: LocalIndex → (LocalService, LocalProcess, OtpNextKey,
                       PermissionMap, OtpMap)
  PermissionMap: RemoteService → (ProcessMap, DomainSet)
    ProcessMap: RemoteProcess → Address
    DomainSet: RemoteDomain
  OtpMap: OtpKey → (RequesterService, RequesterProcess,
                    RecipientService, RecipientProcess,
                    RecipientDomain, RecipientAddress)
```

Figure 5: The NMU's internal nested hash map data structures.

N.3 A method for receiving and storing Sikker permissions.

N.4 A method for checking the permissions of outgoing messages and, if necessary, blocking network access.

To implement high-performance network access, from requirement **N.1**, the NMU implements OS-bypass. As with most other OS-bypass implementations, the NMU allows a process and the NMU to read and write from each others memory space directly without the assistance of the kernel. The NMU's OS-bypass implementation has one major difference compared to other implementations, namely, it uses the memory mapped interface to derive the identity of a communicating process, which fulfills requirement **N.2**. The NMU contains many virtual register sets, upon which, the various processes are able to interact with the NMU. This corresponds to a large physical address space mapped to the NMU. When a new networked process is started, the NMU gives the host's operating system the base address of the register set that the process will use. The NMU contains an internal table that maps register set addresses to process identities. After the process is started, the register set is mapped into the process's memory space and the process is only able to use this register set for interaction with the NMU. The process never tells the NMU its identity, instead, the NMU derives its identity from the memory address used for NMU communication.

The NOS coordinates with every NMU in the network, which reside on each host. The NOS is responsible for creating permissions and distributing them to the proper NMUs. The internal data structures of the NMU have been crafted such that all variable sized data is represented as nested hash maps[2]. Furthermore, the hash mappings and value placements have been optimized to keep the hash maps as small as possible in effort to produce low predictable search times. The elements of the NMU's internal data structures are listed in nested form in Figure 5. These data structures are the NMU's fulfillment of requirement **N.3**. For security reasons, the NMU contains its own memory subsystem that is inaccessible by the host's operating system[3].

To implement the NMU's internal data structures efficiently, the NMU architecture has been designed as a data structure

[2]We consider hash sets the same as hash maps. A hash set is simply a hash map with a zero sized value.

[3]It is possible to use the same memory system as the host processor if the NMU uses digital signatures to verify that the information has not been tampered with.

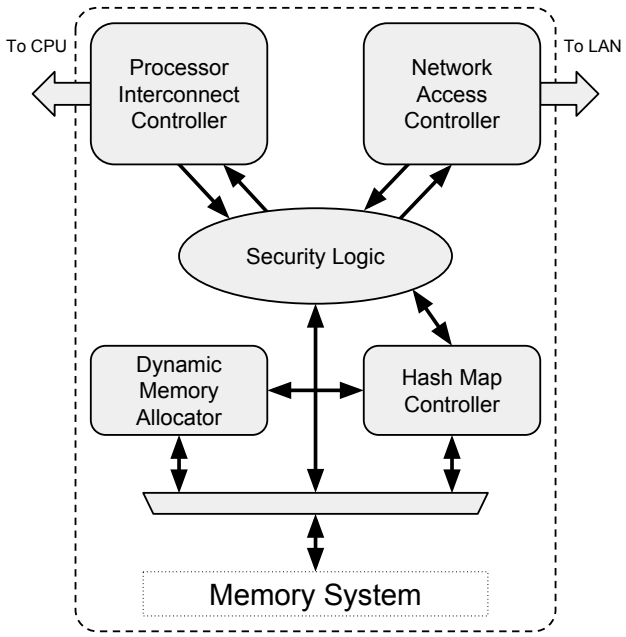

Figure 6: The high-level NMU architecture.

accelerator specifically for managing nested hash maps. As shown in Figure 6, the high-level architecture of the NMU consists of three main blocks: permissions logic, hash map controller, and dynamic memory allocator. The combination of these logic blocks facilitates the management of the internal data structures.

Attached to the memory system of the NMU is the dynamic memory allocator which is a hardware implementation of a coalescing segregated fit free list allocator [10]. This allocator design has a good performance to memory utilization ratio. The allocator allows both the permissions logic and the hash map controller to create, resize, and free dynamically sized blocks of memory. The hash map controller is a hardware implementation of a linear probed open addressing (a.k.a. closed hashed) [41] hash map controller. We chose this particular hash map controller design because it is extremely cache friendly. It connects to the dynamic memory allocator and directly to the memory system. Since the hash map controller handles all hash map operations, the permissions logic simply issues a set of operations for each NMU function.

The NMU's main objective is to efficiently check the permissions of every outgoing message before it enters the network. For each potential message being sent on the network, the permissions logic issues commands to the hash map controller that traverse the nested data structures to ensure that proper permissions exist. If proper permissions do exist, the permissions logic translates the virtual service-oriented network address, consisting of a destination service, process, and domain, into a physical network address. The message is then given to the network access controller to be sent on the network. When proper permissions do not exist, the permissions logic rejects transmission of the message and flags the process with an error code in its corresponding register set. This functionality fulfills requirement **N.4**.

4.2 Operation

In this section we'll walk through the operations the NMU performs and how it traverses and manages the data structures shown in Figure 5. "Local" variables refer to entities resident on the NMU and "Remote" variables refer to entities that exist on other NMUs. It is possible to send messages between processes resident on the same NMU, but we'll keep the "Local" and "Remote" distinctions. When using OTPs, we define the process that generates the OTP as the *requester*, the process that receives the OTP as the *responder*, and the process that receives the message that was sent using the OTP as the *recipient*. Thus, the *requester* sends an OTP and request message to the *responder* and the *responder* uses the OTP to send a response message to the *recipient*. For two-way request-response protocols, the *requester* and *recipient* are the same.

4.2.1 Send

To initiate a standard message send operation, the source process gives the NMU the RemoteService, RemoteProcess, and RemoteDomain of the destination. The NMU derives the sender's LocalIndex which is a simple bit selection from the physical memory address used by the process to communicate with the NMU. Next, the LocalIndex is used as the key for an InfoMap lookup which yields, among other things, the PermissionMap. The NMU then uses the RemoteService to perform a PermissionMap lookup which yields the ProcessMap and DomainSet corresponding to the RemoteService. The NMU now checks that the RemoteProcess exists within the ProcessMap and the RemoteDomain within the DomainSet. If both lookups are successful, the Address that was returned by the ProcessMap lookup is used as the destination physical network address of the message. The message header will contain LocalService and LocalProcess as the message's source and the RemoteService, RemoteProcess, and RemoteDomain as the message's destination. If any lookup during this procedure fails, the NMU will not send the message and will set an error flag in the process's register set.

4.2.2 Receive

When the destination NMU receives the message the destination service, process, and domain have now become the LocalService, LocalProcess, and LocalDomain. Using the LocalService and LocalProcess, the NMU performs an IndexMap lookup which yields the corresponding process's LocalIndex and tells the NMU which register set the message should be placed in.

4.2.3 Send with OTP

When the requester desires to generate and send a message with an attached OTP, on top of specifying the responder as the destination of the message, it must also specify the recipient. The NMU uses the same permission check procedure as in Section 4.2.1 except now it performs two PermissionMap, ProcessMap, DomainSet lookup sequences, one for the responder and one for the recipient. Upon successful lookups, the NMU sends the message just like it did in Section 4.2.1 except that the message header also contains the recipient's information as the OTP.

4.2.4 Receive with OTP

When the responder's NMU receives the message containing the OTP it starts as usual by performing an IndexMap lookup yielding the LocalIndex. It also performs an InfoMap lookup to retrieve the OtpNextKey and OtpMap. The OtpNextKey and the received message are now placed in the corresponding process's register set. The NMU performs a hash map insertion into the OtpMap which maps the OtpNextKey to the OTP information given in the message. The NMU then advances OtpNextKey to the next key and writes it into the memory location where it exists.

4.2.5 Send using OTP

When the responder is ready to send the response message using the OTP, it does not specify the destination in terms of service, process, and domain. Instead, the process gives the NMU the OtpKey it was given during the receive operation. The NMU uses the process's corresponding LocalIndex to retrieve its OtpMap from the InfoMap. The NMU then uses the OtpKey to perform an OtpMap removal operation to retrieve and remove the OTP, which consists of the requester's information as well as the recipient's information. The recipient's information is used as the message destination and the requester's information is also added to the message header so the recipient knows where the message sequence originated from. Since the OTP was removed from the OtpMap during this procedure, the OTP cannot be used again.

5. METHODOLOGY

Since the NMU can be viewed as an extension to the standard NIC architecture, we quantify its performance by measuring the additional latency incurred by performing its operations. The logic of the NMU can be attached to any memory system and the performance of the NMU widely depends on the structure and size of the memory system chosen.

To explore the design space of the NMU and measure its performance, we developed a custom simulator, called *SikkerSim*. The top level of SikkerSim is an implementation of a NOS that manages the permissions of all the NMUs on a network. It does this by creating a permission connectivity graph as shown in Figure 4 and connects a simulated NMU on each simulated host. For each simulated NMU, SikkerSim models the internal logic elements of the NMU as well as various types of memory systems under design consideration. We use SikkerSim to model NMU memory systems spanning from single SRAMs to multi-stage cache hierarchies connected to DRAM. CACTI (32nm process technology) [28] and DRAMSim2 (DDR3 SDRAM) [34] are used in connection with SikkerSim to produce accurate timing results for each case.

For the sake of performance analysis, we chose a memory system design that yields high performance while not incurring excessive cost. This design attaches the NMU logic to a memory system containing two levels of cache and a DRAM main memory. The first cache level (L1) is an 8-way set associative 32 kiB cache. The second cache level (L2) is a 16-way set associative 4 MiB cache. Unlike standard microprocessor cache hierarchies, the NMU operates directly on physical memory addresses and considers all memory as "data". The NMU doesn't need an MMU, TLB, or instruction cache,

Processes per NMU	16
Processes per service	512
Domains per service	256
Service coverage	20%
Process coverage	65%
Domain coverage	25%

Table 3: Connectivity parameters for the synthetic service interaction model.

thus, the NMU's logic connection to the L1 cache is a fast lightweight interface.

5.1 Connectivity Model

SikkerSim contains a synthetic system generator that loads the NOS with hosts, services, processes, and domains based on configurable parameters. The parameters we use for our experimentation are shown in Table 3. As an example, let's consider a system comprised of 131,072 (i.e., 2^{17}) hosts. Under our configuration each host has 16 processes that use the NMU, thus, there are over 2 million processes in the system using Sikker. Since there are 512 processes per service, there are 4,096 total services, each having 256 domains. Each service connects with 819 other services (20% of 4,096) and each service connection is comprised of 333 processes (65% of 512) and 64 domains (25% of 256).

This configuration represents very dense connectivity in a distributed system. In cloud computing environments, there are several very big services but the vast majority of services are small. Small services come from small clients, thus, the inter-process connectivity they require is minimal. The big services that satisfy the requirements of many clients can use the OTP mechanism described in Sections 3.3 and 4.2, thus, they will not need permanent permissions loaded in their NMUs for communicating with their clients.

Large singly-operated data centers (e.g. Facebook) more closely approach our connectivity model as they employ many large services. The majority of modern large-scale web services fit within approximately 1,000 processes, however, they only require connection with approximately 10 other services.

Supercomputers have very little connectivity between services, however, the services themselves can consume enormous portions of the system. Besides services densely connecting with themselves, supercomputer workloads don't exhibit system wide dense connectivity.

5.2 Access Patterns

The data structures of the NMU present abundant spatial locality to the memory system, and depending on the permission access pattern, significant temporal locality can also exist. SikkerSim contains a configurable synthetic permission access pattern that is placed on each simulated NMU. For each permissions check the permission access pattern selects a source and destination. The source specifies which resident process will be accessing the network and the destination specifies a service, process, and domain that the source will be sending the message to.

The worst case access pattern is a uniform random selection across the source and destination possibilities. In this pattern, each permissions check randomly selects a resident process as the source, then randomly selects the destination service, process, and domain from the corresponding source service's permissions. This pattern exhibits no temporal locality in the NMU's memory system.

The best case access pattern is repeatedly choosing the same source and destination. This pattern exhibits full temporal locality in the memory system. While this pattern is unrealistic for long durations, it is realistic for very short durations. A slight variant of this pattern would be repeatedly accessing the same destination service, while switching destination process and/or domain. Similarly, the same source process might be repeatedly accessing the network but choosing a new destination each time.

Since both the worst and best case access patterns are somewhat realistic, we designed the synthetic permission access pattern in SikkerSim to reflect two common attributes that control the temporal locality in a realistic way.

Repeat Groups - The first attribute configures the amount of repeatability at each step of the selection process for the source and destination. There are several aspects that make this realistic in practice. For instance, it is common for a process using the network to interact several times with the network before another process has the chance to or chooses to. This can be caused by CPU thread scheduling or application-level network bursting. Also, it is common for a process to send multiple back-to-back messages to the same destination service or even the same destination service and process and/or service and domain. The result is a higher level of temporal locality simply due to repeated accesses in a particular selection group.

Hot Spot Groups - The second attribute configures the selection distribution when the synthetic permission access pattern chooses a new source and destination. This is used to model hot spots in network traffic. For instance, an application using a SQL database will often also use an in-memory caching service to reduce the load on the SQL database. For this example, the in-memory cache is a hot spot as it is accessed with higher frequency than the SQL database. We allow the selection process to choose using a uniform random distribution or a Gaussian random distribution. The uniform random distribution models network traffic that is irregular and unpredictable while the Gaussian random distribution models network traffic that contains hot spots both in terms of the source and destination with all its components.

Using these controllable attributes, we used SikkerSim's synthetic permission access pattern to create four access patterns that we use to benchmark the performance of the NMU. They are as follows:

- **Uniform Random (UR):** All selections are from a uniform random distribution.
- **Uniform Repeated Random (URR):** Same as UR, except that portions of the selection are re-used a configurable number of times.
- **Gaussian Random (GR):** All selections are from a Gaussian random distribution.
- **Gaussian Repeated Random (GRR):** Same as GR,

except that portions of the selection are re-used a configurable number of times.

6. EVALUATION

6.1 Scalability of SACLs

In this section, we evaluate the scalability of SACLs under the Sikker model. In general, the amount of state needed to represent a set of permissions can be expressed as

$$E = A \times R \tag{1}$$

where E is the total number of ACL entries, A is the number of agents holding permissions, and R is the number of resources being accessed. We compare the NACL methodology to the SACL methodology with the following symbols:

s_t : Total number of services

p_s : Number of processes per service

d_s : Number of domains per service

s_a : Number of accessible services

p_a : Number of accessible processes per service

d_a : Number of accessible domains per service

p_h : Number of processes per host

SACLs have two primary scalability advantages over NACLs. First, SACLs apply permissions directly to services instead of processes. Second, SACLs provide orthogonality between the access to processes and the access to domains. We first evaluate the amount of ACL entries needed in the NOS. For NACLs the number of permission holding agents is equal to the total number of processes in the system. Because NACLs have no knowledge of services, they assume each process has its own domain set. The resulting expression is:

$$N_{nacl} = \underbrace{s_t \times p_s}_{A} \times \underbrace{s_a \times p_a \times d_a}_{R} \tag{2}$$

where N is the number of ACL entries in the NOS. In contrast, the expression for SACLs is:

$$N_{sacl} = \underbrace{s_t}_{A} \times \underbrace{s_a \times (p_a + d_a)}_{R} \tag{3}$$

In Figure 7 the left Y-axis and the solid lines show a comparison between NACLs and SACLs for the storage requirements of the NOS using the connectivity model from Section 5.1. This shows that SACLs maintain savings of over 4 orders of magnitude versus NACLs. For example, if each ACL entry is 4 bytes, and the system size is 131,072 hosts, NACLs requires 146 TB of storage while SACLs only require 5.33 GB.

The amount of storage needed on each host scales differently than the storage required by the NOS. For both NACLs and SACLs, the number of permission holding agents is the number of resident processes. The resulting expression for NACLs is:

$$H_{nacl} = \underbrace{p_h}_{A} \times \underbrace{s_a \times p_a \times d_a}_{R} \tag{4}$$

where H is the number of ACL entries on each host. In contrast, the expression for SACLs is:

$$H_{sacl} = \underbrace{p_h}_{A} \times \underbrace{s_a \times (p_a + d_a)}_{R} \tag{5}$$

In Figure 7 the right Y-axis and the dashed lines show a comparison between NACLs and SACLs for the storage requirements at each host. This shows that SACLs maintain savings of about 2 orders of magnitude over NACLs. For example, if each ACL entry is 4 bytes, and the system size is 131,072 hosts, NACLs requires 1.12 GB of storage while SACLs only require 20.8 MB.

6.2 Latency

This section analyzes the latency incurred in the NMU for checking permissions. Figure 8 shows the mean and 99^{th} percentile latency response of a single permission check for each of the four permission access patterns described in Section 5.2. As expected, the UR and GRR patterns represent the worst and best patterns, however, the mean of the UR pattern is only up to 25% worse than the GRR pattern and both curves flatten out by 32,768 hosts. Even under extreme conditions, the NMU adds negligible latency overhead to network transactions. On a large system with over 2 million processes (131,072 hosts), the mean latency of a realistic access pattern (GRR) is only 41 ns and the 99^{th} percentile latency of the worst case access pattern (UR) is only 66 ns. Relative to the standard permissions checking process, using OTPs incurs the same latency overheads with negligible differences.

6.3 Bandwidth

While predictably low latency is our main metric of performance, bandwidth is also an important metric for high-performance computing. Studies show that the average packet size in common data center traffic is 850 bytes [7]. Given this average packet size, Table 4 shows the throughput of a single NMU logic engine. This shows that a single engine on a very large cluster (131,072 hosts) with a realistic permission access pattern (GRR) can process 166 Gbps on average. Even if we assume the worst case permission access pattern

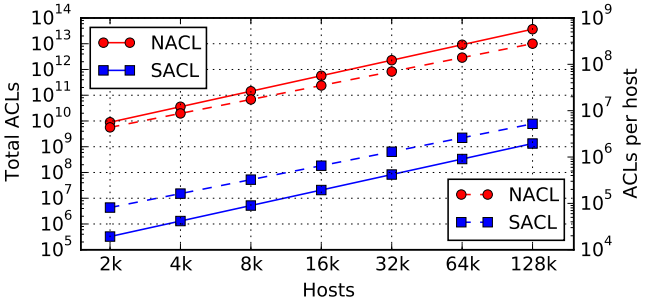

Figure 7: Scalability comparison between NACLs and SACLs. The left Y-axis and solid lines show the storage requirements on the NOS. The right Y-axis and dashed lines show the storage requirements at each host.

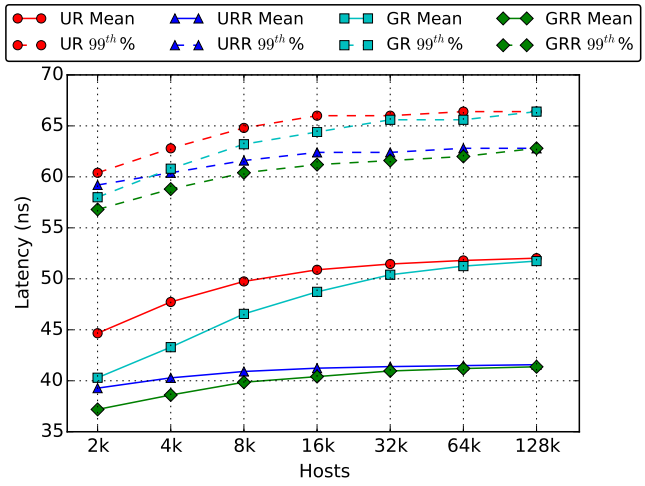

Figure 8: Mean and 99^{th} percentile latency of all four access patterns. Solid lines are mean latencies and dashed lines are 99^{th} percentile latencies.

(UR) and its 99^{th} percentile latency response it can still process 103 Gbps.

	UR		GRR	
	Mean	99^{th}*%ile*	*Mean*	99^{th}*%ile*
Mcps	19.23	15.15	24.39	16.13
Gbps	130.77	103.03	165.85	109.68

Table 4: Bandwidth performance of a single NMU logic engine. *Mcps* is million permission checks per second. *Gbps* is gigabits per second. Average packet size is 850 bytes.

Because the complexity of the NMU is abstracted away by its internal data structures, the complexity of adding multiple logic engines to a single NMU is fairly trivial. Furthermore, the majority of the operations performed in the NMU are read-only operations, which are highly parallelizable. For the operations that require writes (i.e. OTPs), distributing the data structure ownership across multiple engines and using hash-based message steering to the corresponding engine allows near lock-free parallelization. With relatively little effort, an NMU can be built with 4 or more logic engines. Based on the results in Table 4 and degrading performance by 10% to account for potential lock contention, an NMU with 4 logic engines is able to process 55 - 88 million permissions checks per second (i.e. 374 - 598 Gbps).

6.4 Security

The NMU implements all the security and isolation features of Sikker as discussed in Section 3. This includes source and destination authentication, virtual-to-physical network address translation, sender-enforced service-oriented permission checks, and permissions management. Sikker's security model is more straight forward than other approaches because the policies on which it is established are derived directly from the applications themselves, instead of being tied to specific network transport mechanisms. Sikker provides security and isolation mechanisms with far higher granularity than current systems.

Sikker's sender-enforced isolation mechanism removes the ability for denial-of-service attacks between services that don't have permission to each other. This isolation mechanism creates a productive programming environment for developers since they can assume that all permissions checks were performed at the sender. In this environment, developers are able to spend less time protecting their applications from the network and more time developing core application logic.

The Sikker application model uses individual endpoint machines to host the processes of the various services (hence the name *host*). As such, Sikker relies on the host's operating system to provide process-level isolation between the processes resident on that host. In general, Sikker assumes that the various host operating systems within the network are unreliable. For this reason, the NMU was designed to be explicitly controlled by the NOS rather than individual host operating systems.

In the event that a host's operating system is exploited by a resident process, the process might be able to assume any of the permissions that have been given to *all* processes on that host. This is a large improvement over current systems that utilize the host operating systems for security (e.g., hypervisor-based security and isolation). In those systems, an exploited operating system might be given access to anything in the entire network, not just the permissions resident on that host. In Sikker, if a host's operating system cannot be deemed reliable enough provide process-level isolation, it is recommended to co-locate processes only where an attack would not prove detrimental if one resident process gained access to another resident process's permissions.

7. RELATED WORK

7.1 Supercomputers

For the sake of performance, modern supercomputers employ minimal security and isolation mechanisms. For isolation, some fabrics use coarse-grained network partitioning schemes that are efficient at completely isolating applications from each other but they don't provide a mechanism for controlled interaction between applications. This is especially problematic if the system offers shared services, such as a distributed file system (e.g., Lustre [8]).

Some high-performance interconnects, namely InfiniBand, employ mechanisms for secret key verification where the receiving network interface is able to drop packets that do not present the proper access key that corresponds to the requested resource [17]. While this scheme provides a mechanism for coarse-grained security, it does not provide network isolation nor does it provide fine-grained security to cover the application's security requirements. As a result, the endpoints are susceptible to malicious and accidental denial-of-service attacks and they still have to implement the required fine-grained security checks in software.

Current research in the space of supercomputer multi-tenancy focuses on resource utilization and fairness and makes little effort to provide security and isolation in the face of malicious behavior. These proposals [37, 21, 22, 46, 9], while succeeding in their defined goals, do not provide secure supercomputing systems in the presence of multi-tenancy. Fur-

thermore, none of these proposals provide an architecture on which large-scale service-oriented applications can be built with scalability. Given these systems, supercomputers are still only useful for the well-behaved scientific computing community.

7.2 Cloud Computing

In contrast to supercomputers, cloud computing facilities (e.g., Amazon Web Services [2], Microsoft Azure [25], Google Cloud Platform [15], Heroku [35], Joyent [19]) are faced with the most malicious of tenants. These facilities run applications from many thousands of customers simultaneously, some as small as one virtual machine and others large enough to utilize thousands of servers. These facilities must provide the highest level of security and isolation in order to protect their clients from each other. Furthermore, these facilities often have large sharable services that get used by their tenants for storage, caching, messaging, load balancing, etc. These services also need to be protected from client abuse.

Network isolation mechanisms found in modern cloud computing facilities are network partitioning schemes both physical and virtual (e.g., VLAN [33], VXLAN [23], NVGRE [40], etc.). These partitioning schemes are successful at completely isolating applications from each other, but just like the partitioning schemes found in supercomputers, they don't provide a mechanism for controlled interaction between partitions. In efforts to bridge partitions, network virtualization software like OpenStack Neutron [30] and VMware NSX [44] create virtualized switches (e.g., Open vSwitch [31]) that use NACLs to control the inter-partition interactions.

Current research in the space of cloud computing multi-tenancy uses hypervisor-based pre-network processing to implement various types of security and isolation. While these proposals [20, 16, 39, 38, 6, 32] achieve their desired goals of providing fair network resource sharing, they significantly increase message latency and CPU utilization and *still* don't provide fine-grained security and isolation. These proposals are often developed and tested on network bandwidths an order of magnitude lower than the bandwidths achieved on supercomputers (10 Gbps vs 100 Gbps) and may not be feasible at supercomputer bandwidths.

Due to the rate of increasingly more bandwidth in the data center and the plateau of CPU performance, the cost of virtual switching is outrunning the abilities of the CPUs on which the hypervisor executes. A recent study [27] shows that in 2005 a Xeon-class server with 1 Gbps Ethernet dedicated about 15% of its cycles to networking overhead. By 2010, with Nehalem Xeons, 10 Gbps Ethernet, and the move to virtual switching the overhead rose to 25%. According to the study, the overhead of Haswell Xeons matched with 25 Gbps is 33% and the overhead of future Skylake Xeons matched with 50 Gbps Ethernet will be 45%.

It is well known that cloud computing environments impose high network overheads and unpredictable performance on their clients [45, 13]. While we do not claim that all of these poor results are related to security and isolation, it is evident that modern network virtualization and hypervisor-based techniques cause significant overhead. A recent study [43] shows that two virtual machines communicating on the same host should expect 25-75 µs of round-trip latency. Similarly, a virtual machine communicating with a native operating system connected to the same 10 Gbps physical switch should expect 35-75 µs of round-trip latency. The latency is significantly worse if the communication is forced to go through an intermediate host containing a virtual router in order to cross the boundary between virtualized networks, as is done in OpenStack Neutron [30].

8. CONCLUSION

In this paper we have introduced a new distributed system architecture, called Sikker, with an explicit security and isolation model designed for large-scale distributed applications that run in data centers, cloud computing facilities, and supercomputers. Sikker is designed to be a high performance and scalable solution to enforce the permissions of the complex interactions of modern distributed applications. Sikker's service-oriented application model is an intuitive and effective alternative to network-derived ACL systems as it was derived directly from the interactions and structure of modern large-scale applications.

We've presented the Network Management Unit (NMU), a network interface controller that efficiently enforces the permissions scheme of Sikker. Working under the direction of a network operating system, the NMU provides network isolation through enforcing permissions at the sender and provides security through its inherent implementation of the principle of least privilege as well as source and destination authentication. Even when paired with the highest performing interconnection networks, the NMU induces negligible overhead for network transactions and is able to scale to future systems with even higher performance.

Sikker and the NMU enable a new generation of distributed systems performing like supercomputers while operating with inherent service-oriented security and isolation. This new generation of computing supports large-scale multi-tenant computing platforms where system architects and application developers are able to access remote data quickly, spend less time writing tedious and error-pone security checks, and spend more time developing core application logic.

9. REFERENCES

[1] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE, 2010, pp. 83–87.

[2] Amazon. Amazon web services (aws). [Online]. Available: http://aws.amazon.com

[3] ——. High performance computing. [Online]. Available: https://aws.amazon.com/hpc/

[4] ——. Simple storage service (s3). [Online]. Available: https://aws.amazon.com/s3/

[5] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The percs high-performance interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. IEEE, 2010, pp. 75–82.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 242–253, 2011.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[8] P. J. Braam, "The lustre storage architecture," 2004.

[9] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on hpc systems with node sharing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 37.

[10] R. Bryant and O. David Richard, *Computer systems: a programmer's perspective*. Prentice Hall, 2003.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[12] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The ibm blue gene/q interconnection network and message unit," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–10.

[13] J. Ciancutti. (2010, December) 5 lessons we've learned using aws. [Online]. Available: http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

[14] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable hpc system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 103.

[15] Google. Google cloud platform. [Online]. Available: http://cloud.google.com

[16] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International Conference*. ACM, 2010, p. 15.

[17] Infiniband Trade Association, "Infiniband architecture specification," 2000.

[18] J. Jackson. (2014, July) Ibm aims to disrupt supercomputing market with cloud enticements. [Online]. Available: http://www.pcworld.com/article/2457580/ibm-aims-to-disrupt-supercomputing-market-with-cloud-enticements.html

[19] Joyent. High-performance cloud computing. [Online]. Available: http://www.joyent.com

[20] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler, "Softudc: A software-based data center for utility computing," *Computer*, no. 11, pp. 38–46, 2004.

[21] M. C. Kurt and G. Agrawal, "Disc: a domain-interaction based programming model with support for heterogeneous execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 869–880.

[22] H. Liu and B. He, "Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 970–981.

[23] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks," *draftmahalingam-dutt-dcops-vxlan-01. txt*, 2012.

[24] Mellanox Technologies. (2015) Infiniband performance. [Online]. Available: http://www.mellanox.com

[25] Microsoft. Azure: Microsoft's cloud platform. [Online]. Available: http://azure.microsoft.com

[26] ——. Azure search - search-as-a-service for web and mobile app development. [Online]. Available: https://azure.microsoft.com/en-us/services/search/

[27] T. Morgan. Broadcom goes end to end with 25g ethernet. [Online]. Available: http://www.nextplatform.com/2015/07/27/broadcom-goes-end-to-end-with-25g-ethernet/

[28] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[29] Netflix. Netflix cloud architecture. [Online]. Available: http://www.slideshare.net/adrianco/netflix-velocity-conference-2011

[30] OpenStack Foundation. Openstack neutron. [Online]. Available: https://wiki.openstack.org/wiki/Neutron

[31] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[32] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 187–198.

[33] V. Rajaravivarma, "Virtual local area network technology and applications," in *Southeastern Symposium on System Theory*. IEEE Computer Society, 1997, pp. 49–49.

[34] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[35] Salesforce. Heroku. [Online]. Available: http://www.heroku.com

[36] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[37] O. Sarood, A. Langer, A. Gupta, and L. Kale, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 807–818.

[38] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network." in *NSDI*, 2011.

[39] P. Soares, J. Santos, N. Tolia, D. Guedes, and Y. Turner, "Gatekeeper: Distributed rate control for virtualized datacenters," 2010.

[40] M. Sridharan, K. Duda, I. Ganga, A. Greenberg, G. Lin, M. Pearson, P. Thaler, C. Tumuluri, N. Venkataramiah, and Y. Wang, "Nvgre: Network virtualization using generic routing encapsulation," *IETF draft*, 2011.

[41] A. M. Tenenbaum, *Data structures using C*. Pearson Education India, 1990.

[42] Twitter. Finagle: A protocol-agnostic rpc system. [Online]. Available: https://blog.twitter.com/2011/finagle-a-protocol-agnostic-rpc-system

[43] VMware. Network i/o latency on vsphere 5, performance study. [Online]. Available: http://www.vmware.com/files/pdf/techpaper/network-io-latency-perf-vsphere5.pdf

[44] ——. Nsx. [Online]. Available: http://www.vmware.com/products/nsx

[45] F. Xu, F. Liu, H. Jin, and A. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, Jan 2014.

[46] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 60.