Logical Partitions on Many-Core Platforms

Ramya Jayaram Masti, Claudio Marforio, Kari Kostiainen,
Claudio Soriente, Srdjan Capkun
Institute of Information Security, ETH Zurich
firstname.lastname@inf.ethz.ch

ABSTRACT

Cloud platforms that use logical partitions to allocate dedicated resources to VMs can benefit from small and therefore secure hypervisors. Many-core platforms, with their abundant resources, are an attractive basis to create and deploy logical partitions on a large scale. However, many-core platforms are designed for efficient cross-core data sharing rather than isolation, which is a key requirement for logical partitions. Typically, logical partitions leverage hardware virtualization extensions that require complex CPU core enhancements. These extensions are not optimal for many-core platforms, where it is preferable to keep the cores as simple as possible.

In this paper, we show that a simple address-space isolation mechanism, that can be implemented in the Network-on-Chip of the many-core processor, is sufficient to enable logical partitions. We implement the proposed change for the Intel Single-Chip Cloud Computer (SCC). We also design a cloud architecture that relies on a small and disengaged hypervisor for the security-enhanced Intel SCC. Our prototype hypervisor is 3.4K LOC which is comparable to the smallest hypervisors available today. Furthermore, virtual machines execute bare-metal avoiding runtime interaction with the hypervisor and virtualization overhead.

1. INTRODUCTION

A logical partition is a subset of the physical system resources that can run an operating system independently of the rest of the system [19]. Logical partitions are widely used in high-assurance virtualized environments such as formally-verified separation kernels [13, 26] and commercial hypervisor deployments [1, 2, 19]. For example, IBM utilizes logical partitions in its Infrastructure as a Service (IaaS) cloud [4]. Hypervisors that use logical partitions, provide dedicated resources to the hosted operating systems and, therefore, benefit from lightweight resource management and a small Trusted Computing Base (TCB). Logical partitions also allow to reduce the runtime attack surface by minimizing the interaction between the hypervisor and the hosted operating systems — a technique called hypervisor disengagement [39]. Other benefits of logical partitions include improved performance [15] and efficient nested virtualization [36].

A many-core processor consists of a large number of simple, energy-efficient cores integrated on a single processor die. Due to their abundant resources, many-core platforms may be used, in principle, for deploying systems that scale up to hundreds, or even thousands [7], of logical partitions. However, the available many-core systems [7, 21, 22, 41] are designed for high-performance computing applications and allow data sharing across cores. Consequently, such many-core architectures are not tailored to support efficient isolation which is a mandatory requirement for logical partitions.

In this paper, we investigate the feasibility of enabling logical partitions on many-core processors, in order to integrate the processors into IaaS clouds. Motivated by the scalability benefits of simple cores, we discard the option of relying on hardware virtualization extensions. We show that, without virtualization extensions, logical partitions can be facilitated by a simple address-space isolation mechanism, implemented in the Network-on-Chip of the many-core processor. This simple hardware enhancement does not affect the processor core's complexity and therefore supports many-core processor scalability.

We use the Intel SCC many-core architecture [22] as a case study and demonstrate the feasibility of our solution with an emulated software prototype. With the proposed hardware changes in place, we design a simple, disengaged hypervisor for many-core processors and integrate it into an IaaS cloud. Our prototype hypervisor has an implementation size of only 3.4K LOC that is comparable to the smallest hypervisors available today [46]. In contrast to solutions like NoHype [39], our hypervisor does not rely on hardware virtualization extensions. Furthermore, similar to [15, 36], our solution allows bare-metal execution of VMs, hence eliminating virtualization overhead.

This work demonstrates that many-core processors are an attractive basis for implementing secure cloud computing services. To summarize, we make the following contributions:

1. We show that on many-core platforms logical partitions can be implemented with a simple hardware enhancement, avoiding the need for hardware virtualization extensions. We use the Intel SCC architecture as a case study and propose minor hardware enhancements that enable secure partitions.
2. We design and implement a complete IaaS architecture that leverages the security-enhanced Intel SCC

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3682-6/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818026

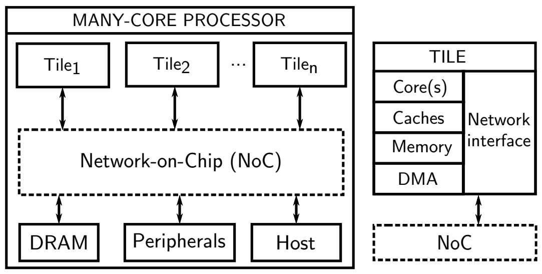


Figure 1: **Many-core architecture overview.** Most many-core processors are organized in tiles that are connected to memory and peripherals over a Network-on-Chip. Each tile has one or more cores, network interface and optionally some local memory and a DMA engine. Each processor could include a host interface if it is a co-processor.

platform. The architecture includes a disengaged hypervisor with a TCB size comparable to the smallest known hypervisors.

3. We evaluate the performance of our architecture and demonstrate its suitability for practical cloud deployments.

The rest of this paper is organized as follows. In Section 2 we provide background information on many-core architectures. Section 3 describes hypervisor design alternatives and explains the goals of this work. Section 4 presents our security enhancements to the Intel SCC platform and an emulated implementation. In Section 5 we describe an IaaS cloud architecture, its implementation and evaluation. In Section 6 we discuss deployment properties. Section 7 reviews related work and we conclude in Section 8.

2. BACKGROUND

2.1 Many-Core Architectures

A many-core platform consists of a large number of cores (hundreds or even thousands [7]) integrated into a single processor chip. Figure 1 shows the general architecture of a many-core platform. Many-core processors are typically organized as tiles that can access memory and peripherals over an interconnect. A network interface connects the components on each tile to the Network-on-Chip (NoC). Each tile contains a set of cores and, optionally, some local memory and a DMA engine. All system resources, including peripherals, are accessible via memory-mapped I/O. If the platform is designed to be a co-processor, it may optionally incorporate a dedicated memory module (DRAM) and an Ethernet card that is separate from the host platform. The number of cores, their underlying instruction set (e.g., x86, RISC), and the type of interconnect (e.g., ring, mesh) vary across platforms.

Many-core platforms can be broadly classified by their adherence to the symmetric or asymmetric multi-processor specifications (SMP or AMP). SMP systems, such as Intel Xeon Phi [21], include hardware support that enables all cores in the system to be controlled by a single operating system. SMP systems achieve this by implementing features like a centralized reset for cores, the ability to access the entire system memory from any core, and cache-coherence.

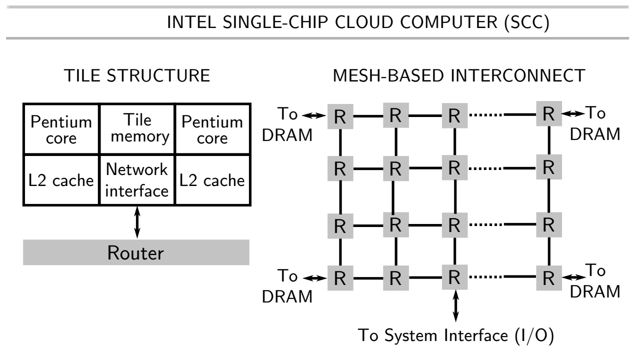


Figure 2: **Intel SCC architecture overview.** The processor has 24 tiles and each tile hosts two cores. Each tile is connected to a mesh Network-on-Chip (NoC) via a network interfaces and a router. Cores access shared memory (DRAM) on the processor and peripherals, via the NoC.

Asymmetric multi-core processors, such as Intel SCC [22], are in contrast designed to execute multiple operating systems on the same platform.

While most current many-core platforms do not support virtualization extensions, there are also examples of SMP architectures that provide such functionality partially. For example, the Tilera TILE-GX [41] processors include memory virtualization support but lack CPU virtualization. Since future many-core platforms are likely to scale to hundreds or even thousands of cores, it is desirable to keep the cores as simple as possible. Solutions where operating systems run directly on the hardware, without explicit virtualization mechanisms also have performance benefits [15,36].

We, therefore, focus on the Intel SCC platform which is an AMP architecture capable of running multiple operating systems directly on its cores.

2.2 Intel SCC

Figure 2 shows an overview of the Intel SCC architecture. Intel SCC is a co-processor that is connected to a host platform. The processor has 48 cores arranged in 24 tiles. Each tile contains two Pentium P54C cores enhanced with an L2 cache and special instructions to use the local on-tile memory. Cores can be dynamically turned on and off depending on current computing load. Each tile also has some local memory that is shared between the cores called the Message Passing Buffer (MPB) and an on-tile network interface that connects the tile to a router. Each core is capable of running a separate operating system independent of the other cores. No two cores on the system share any cache and the platform does not implement cache-coherence.

The routers at each tile are connected to a mesh network which allows cores to access off-tile memory (DRAM) and I/O peripherals. Routers at the corners of the mesh network are connected to memory controllers and each memory controller is connected to a DRAM memory element. Each core uses the DRAM module attached to the memory controller at the closest NoC corner.

All resources in the Intel SCC are memory-mapped and can be configured through a set of Look-Up Tables (LUTs) in the network interface of each tile. Each LUT entry translates a range of physical addresses to system-wide addresses, as shown in Figure 3. A system-wide address can point either

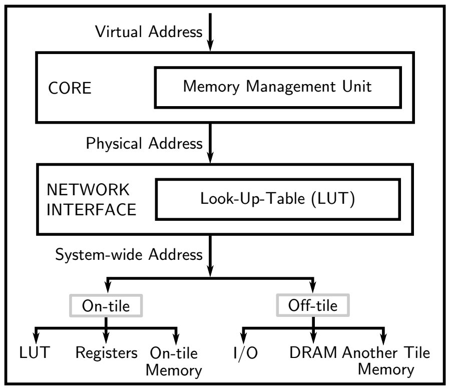


Figure 3: **Intel SCC address translation.** On each core, an MMU translates virtual addresses to physical addresses and a look-up-table (LUT) translates physical addresses to system-wide addresses.

to an on-tile or to an off-tile resource. Cores access external DRAM memory, memory-mapped peripherals, and memory on other tiles, using off-tile memory addresses. Using an on-tile memory address, a core can access on-tile memory, registers, and its own LUT.

3. DESIGN ALTERNATIVES

3.1 Hypervisor Designs

The typical way to realize logical partitions is using a hypervisor that is a privileged software layer that manages the execution of multiple operating systems on shared physical resources. The hypervisor is responsible for the isolation of the operating systems, and thus it is part of the system TCB. A common approach to reduce the risk of TCB compromise is to minimize the implementation size of the hypervisor and its runtime attack surface. We call these properties *small footprint* (P1) and *reduced interaction* (P2).

The hypervisor footprint is typically measured in terms of the source lines of code (LOC). The smallest hypervisors available today leverage hardware virtualization extensions and have a footprint of around 4K LOC [46]. In contrast, hypervisors that implement all the virtualization and management functionality in software, are typically above 100K LOC [40].

The runtime attack surface mainly depends on the interaction of the hypervisor with the untrusted components in the system and in particular with the operating systems running in the logical partitions. While a typical hypervisor interacts with the operating systems during their launch, operation and shutdown, a disengaged hypervisor limits its interaction with the OS to the time of the OS launch and shutdown [39]. Such reduced interaction is beneficial for security, as the OS-hypervisor interfaces tend to contain bugs and may allow hypervisor exploitation [31].

Below, we discuss different hypervisor designs and analyze their ability to achieve properties P1 and P2. Hypervisors can be broadly classified into three categories, as illustrated in Figure 4.

Traditional hypervisor. A single hypervisor instance manages all the cores in the platform and interacts with the operating systems running in the logical partitions (Figure 4.a).

The footprint of the hypervisor depends on its virtualization technique and available hardware virtualization extensions. If virtualization and OS management are implemented in software (e.g., para-virtualization in Xen or binary translation in VMware), the hypervisor footprint is typically above 100K LOC [40]. Hardware virtualization extensions enable hypervisors with smaller footprints [38, 46]. Typical virtualization extensions include cores with an additional de-privileged execution mode and hardware-assisted second-level address translation. For example, Intel x86 processors provide a guest mode for execution of the OS and Extended Page Tables (EPT) to translate second-level addresses in the memory management unit (MMU). On platforms with hardware virtualization extensions, the OS runs in the de-privileged mode and manages its memory without hypervisor involvement. These mechanisms simplify hypervisor functionality and hence reduce its size.

Despite the small footprint (P1), traditional hypervisors require frequent interaction with the hosted operating systems and do not provide reduced interaction (P2). For example, since each core is shared between the OS and the hypervisor, each external I/O interrupt causes a context switch from the OS to the hypervisor.

Distributed hypervisor. In a distributed design [10, 32], a separate hypervisor instance manages each core or a subset of the available cores (Figure 4.b). Similar to traditional hypervisors, the footprint of a distributed hypervisor depends on the available virtualization support in the processor. Since cores are shared between the hypervisor and the hosted operating systems, reduced hypervisor-OS interaction is not possible. Therefore, distributed hypervisors can provide property P1 but not property P2.

Centralized hypervisor. A centralized hypervisor [15, 36, 39] runs on a dedicated core and manages operating systems that are executed on the other cores of the platform (Figure 4.c). Current centralized hypervisors leverage virtualization extensions. A centralized design can limit the runtime interaction with the hosted operating systems for improved security [39] and performance [15, 36]. For example, to launch an OS, NoHype [39] starts a small setup utility in the privileged execution mode on the target core. The setup utility configures the memory mappings for the hardware-assisted second-level address translation. Then, the hypervisor starts the OS execution in the de-privileged guest mode. A centralized hypervisor can provide both properties P1 and P2. This approach can also improve performance, as the hosted operating systems run directly on the hardware without runtime virtualization overhead.

3.2 Goal: Logical Partitions with Simple Cores

Our analysis shows that only a centralized hypervisor design can provide small footprint (P1) and reduced interaction (P2). Our goal in this paper is to realize logical partitions with both security properties on many-core platforms.

The known techniques to implement small and disengaged hypervisors require hardware virtualization extensions (e.g., NoHype [39]). The majority of current many-core platforms do not provide virtualization support and adding such extensions to the cores has scalability disadvantages. Furthermore, virtualization itself increases the runtime overhead of VMs [15, 36]. For these reasons, we focus on alternatives that enable logical partitions on many-core platforms without requiring processor virtualization.

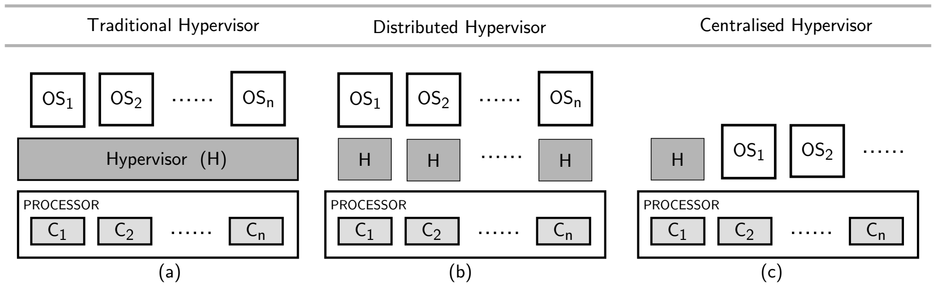


Figure 4: **Hypervisor design alternatives.** A hypervisor can be organized as (a) a single instance that manages all cores, (b) a distributed kernel that treats the underlying platform as a distributed system, or (c) a centralized kernel that allows each OS to execute directly on the hardware.

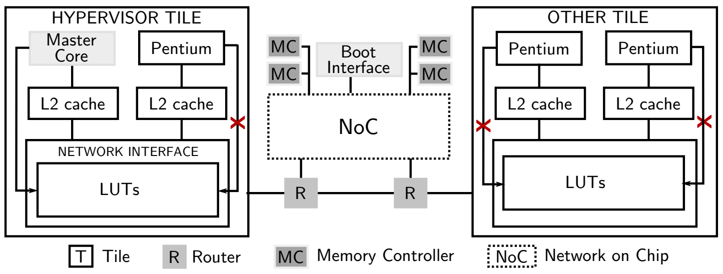


Figure 5: **Security-enhanced Intel SCC.** We propose two security enhancements. First, a LUT protection mechanism that restricts access to LUTs to a single core. The master core can configure all LUTs on the platform through the Network-on-Chip (NoC). Second, we add a boot interface to enable the SCC to operate as a stand-alone processor.

4. LOGICAL PARTITIONS IN MANY-CORE PROCESSORS

Hardware virtualization extensions were designed for systems where each core is shared between the hypervisor and an operating system. Processors with virtualization extensions provide a de-privileged execution mode for the operating systems and at least two levels of address translation. Similar techniques are not required to implement logical partitions using a centralized hypervisor on many-core systems, because the hypervisor and the operating systems can run on separate cores. In logical partitioning schemes that use a centralized hypervisor, all the cores use only one of the two privilege modes (hypervisor/OS), and therefore do not need to support both.

We observe that if the hypervisor and the operating systems run on different cores, a small and disengaged hypervisor can be realized when the underlying processor incorporates mechanisms that confine each core (and hence the OS running on top) to its own set of resources. Since all resources in many-core processors are memory-mapped, this essentially reduces to restricting access of each core to a set of system addresses. We refer to this mechanism as *address-space isolation*.

On many-core systems, address-space isolation can be realized without introducing additional functionality into the processor cores. For example, the Network-on-Chip (NoC) that connects the cores to the system main memory can be enhanced to enforce address-space isolation. NoC-based access control techniques have been studied in the context of embedded systems to isolate applications (e.g., [17, 34]).

We use the Intel SCC architecture as a case study and show that address-space isolation can be realized with very simple enhancements to the processor NoC. We chose the Intel SCC because it has simple cores that are capable of running OS instances independently and its memory addressing scheme provides a good basis for implementing address-space isolation.

4.1 Security-enhanced Intel SCC

To support address-space isolation, and thus secure logical partitions, we propose the following two, simple changes to the Intel SCC architecture:

- **LUT protection.** The first enhancement is a LUT protection mechanism that allows modifications of all LUTs only from a single *master core*. In practice, this can be implemented by restricting access to LUTs from all but one core, as illustrated in Figure 5. The master core can configure all LUTs in the platform over the NoC. At runtime, the (read-only) LUT configuration at each core enforces address-space isolation for the operating system running on top of it. This modification allows to enforce access control not only to memory, but also to peripherals and to other cores.

- **Stand-alone boot.** The second enhancement is a bootable ROM module (boot interface) that allows the Intel SCC to be used as a stand-alone processor. This modification obviates the need for a host platform and hence the need to include the host OS in the system TCB. The proposed modification is inline with the recent trend of stand-alone many-core processors, instead of their use as co-processors [20].

The required hardware modifications are small and do not affect the scalability of the platform. In fact, the modifica-

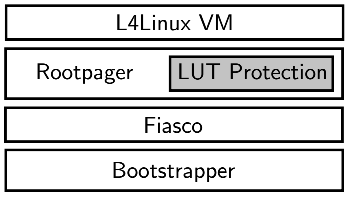


Figure 6: **SCC enhancement emulation.** We use a customized Fiasco microkernel running on each core of the SCC platform to emulate the LUT protection features.

tions we propose do not increase the complexity of the Intel SCC cores at all.

In the Intel SCC architecture, each tile contains two cores that must share an on-tile memory region (for registers) due to the granularity of the addressing mechanism and the available on-tile memory. Consequently, our simple hardware modification supports logical partitions at the granularity of a tile (two cores). To enable per-core partitions, the Intel SCC architecture should be enhanced to completely separate the address space of the two cores on each tile. In this paper, we focus on demonstrating the concept of NoC-based address-space isolation and realize per-tile partitions.

4.2 Emulated Implementation

We provide an implementation where we emulate the proposed hardware enhancements. The purpose of this implementation is to demonstrate that we can provide address-space isolation on the Intel SCC platform simply by controlling the access of the cores to their LUTs.

Our setup consists of an Intel Xeon server that connects to the Intel SCC co-processor via a PCI Express (PCIe) interface. The host runs Ubuntu 10.10 and includes a driver for the Intel SCC. The driver is used to enable user-space access to individual cores (e.g., to reset them) and to the DRAM on the Intel SCC to load applications from the host.

To emulate the boot functionality, we use the host to load code on the master core (core 0) of Intel SCC and start it. To realize the LUT protection, our prototype uses a Fiasco microkernel [33]. Figure 6 shows the components in the kernel implementation. A Bootstrapper module starts the Fiasco microkernel when a core is powered on. We modified the Rootpager module to enforce that the OS running on top of the microkernel cannot modify LUTs, as the Rootpager mediates all address space requests in the Fiasco architecture.

To demonstrate address-space isolation, we run L4Linux [42] (a Linux flavor compatible with Fiasco) on top of the enhanced microkernel. On cores other than the master core, the L4Linux instances were not able to access system resources (memory, cores, I/O devices) other than the ones allocated for them, and importantly they were not able to change their own LUT configuration.

5. MANY-CORE CLOUD

In this section, we describe a complete IaaS architecture that leverages the security-enhanced Intel SCC platform. Figure 7 illustrates the main components of a typical IaaS cloud infrastructure. Customers log in via a *management service* which also allows them to start, stop, or delete their virtual machines. Each VM runs an independent operating system. A *storage service* provides persistent storage for customer VMs and their data. A VM is assigned to, and

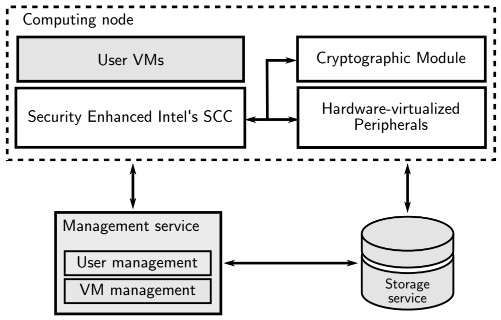


Figure 7: **IaaS cloud infrastructure.** A typical IaaS infrastructure consists of three main components: a computing node, a management service and a storage service. Our computing node is a security-enhanced Intel SCC, connected to a cryptographic module and hardware-virtualized peripherals. We consider the parts shown in gray untrusted.

runs on, one of many *computing nodes.* A hypervisor on the computing node manages VMs and ensures isolation.

The computing node has access to an hardware *cryptographic module* that securely stores the private key of an asymmetric key pair. The corresponding public key is certified by the cloud provider or by a trusted authority. The cryptographic module decrypts VM images encrypted under its public key, before the VM is loaded on a core of the computing node. The decryption interface is available to the hypervisor of the computing node and it is disconnected from the other cloud components (e.g., the management and the storage services). Hardware cryptographic modules, such as IBM cryptographic co-processors [8], are commonly used in cloud platforms.

In our architecture, the computing node is based on the security-enhanced Intel SCC platform and equipped with hardware-virtualized peripherals. For example, a network card with the Intel SR-IOV technology [23] can be configured to create multiple virtual functions. Each VM is allocated a separate virtual function to access the network and storage on a networked file-system. The many-core processor also has a chipset (e.g., one supporting the Intel VT-d extensions [24]) that can be used to re-route interrupts from the hardware-virtualized network card to the associated cores on the many-core processor.

5.1 Security Goals

We consider two types of threats. In *cross-VM attacks* a malicious VM tries to access the data of a victim VM running on the same platform. In *cloud platform attacks*, compromised components of the cloud architecture (e.g., the management service or the storage service) try to access the data of the victim VM. Our goal is to design a cloud architecture that withstands both types of attacks. In Figure 7, we highlight the cloud components that we consider untrusted.

We exclude hardware attacks and consider the underlying hardware trustworthy. Similarly, we exclude denial-of-service attacks because the solutions to them are complementary to our work (see Section 6 for further discussion). Side-channel attacks are discussed in Section 5.3.

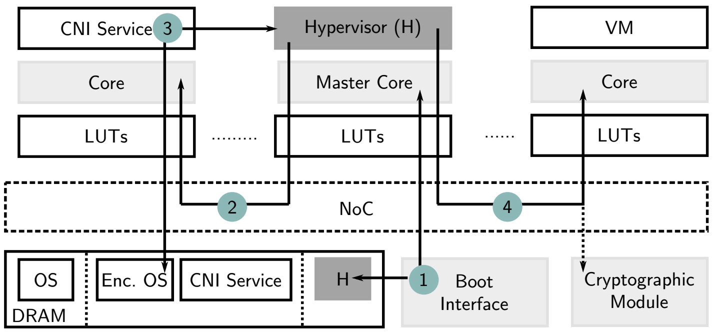


Figure 8: **Cloud architecture and operation.** The boot interface starts the hypervisor on master core. The hypervisor starts the CNI service which fetches VM images from the storage service. The hypervisor sets up new partitions, uses a cryptographic module to decrypt VM images and loads them for execution.

5.2 Architecture and Operation

Figure 8 depicts a cloud architecture based on the security-enhanced Intel SCC and its operations.

The hypervisor boots on the master core at power on (Step 1) and resets the LUTs of all cores. To reduce the TCB size we follow a technique known as hypervisor factorization [18, 43] and run non-critical services of the computing node in separate partitions. For example, after boot, the hypervisor starts a *Compute Node Interface* (CNI) service in a separate logical partition (Step 2). This service fetches VMs over the network to run them on the computing node but is not part of the TCB as it may include complex networking libraries. The hypervisor may also start other services, such as those for resource accounting, in their own partitions.

Customers encrypt their VM images under the public key of the cryptographic module and upload them to the cloud. Customers may also inject keys into their encrypted VM images to enable authentication upon login (e.g., via SSH).

When the VM is scheduled for execution, the CNI service fetches the encrypted VM from the storage service and loads it to memory (Step 3). The hypervisor picks a free core and configures its LUT to enable access to an exclusive memory region and instances of virtualized peripherals (e.g., a virtual network card). The hypervisor copies the encrypted VM image to a memory region shared exclusively with the cryptographic module. The cryptographic module decrypts the VM image using the private key and the hypervisor loads it into the memory region that was allocated for the new partition. Finally, the hypervisor resets that core to start the VM's execution (Step 4).

At VM shutdown, the hypervisor clears the memory assigned to that VM, so that it can be securely re-allocated to other VMs, and resets the LUT of the freed core. If a VM must save sensitive data, it can encrypt the data under its own key before shutting down. The ciphertext is stored on the networked file-system and will be available upon the next execution of the VM.

Memory and I/O management. In the Intel SCC, each LUT entry corresponds to an address space region of 16 MB. Therefore, the hypervisor allocates DRAM memory in chunks of 16 MB. The chunks assigned to each VM need not be contiguous and memory chunks assigned to a VM can be easily re-assigned to other VMs by simply configuring the LUTs accordingly.

The LUT mechanism is also used to manage virtualized I/O devices. Such virtualized devices usually expose a set of *virtual instances* (e.g., [16]). Each virtual instance usually consists of a set of memory-mapped registers that can be exposed to individual VMs by configuring their LUTs appropriately. The interrupt management features available on modern chipsets can then be used to route interrupts from the I/O device to individual cores.

5.3 Security Analysis

The TCB of our architecture consists only of the hypervisor and the underlying hardware.

To protect against cross-VM attacks, the hypervisor confines each VM to its allocated resources. In particular, the hypervisor configures the LUTs so that a VM cannot access a DRAM memory region, a virtual peripheral instance, or on-tile memory of another partition. Furthermore, the hypervisor clears the contents of memory regions before their re-allocation. Only the hypervisor, running on the master core, can modify the configuration of the address-space isolation mechanism.

To protect against cloud platform attacks, VM images at rest are encrypted under the public key of the cryptographic module. Address-space isolation ensures that the hypervisor is the only component on the computing node that can access the cryptographic module through shared memory.

An adversary that manages to compromise the CNI service or the storage service can only access encrypted VM images. If the adversary compromises the management service, he may impersonate a customer and, for example, substitute the customer's VM with one of his choice. Both attacks result in a denial-of-service and are out of our scope. Even if a user's credentials to log into the management service are leaked, the adversary cannot access any of the VMs belonging to that user, as long as those VMs require the user's private key for authentication.

Side-channels are a relevant attack vector in any system where multiple VMs run on the same platform. Side-channel attacks that extract fine-grained information, such as a cryptographic key, from a victim VM using shared cache [44, 45] and memory deduplication [25] have been demonstrated. These attacks are not applicable in our architecture, as partitions share no cache and our hypervisor does not implement deduplication. More coarse-grained attacks that leak VM co-residency based on IP addresses [35] are applicable also in

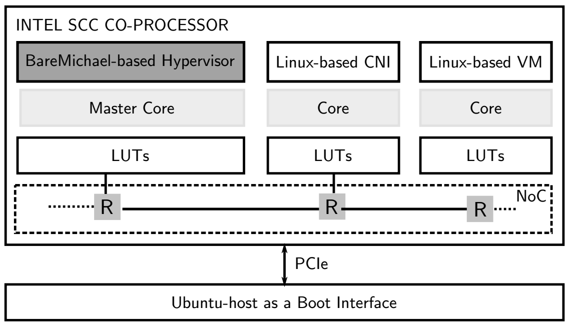


Figure 9: **Compute node implementation.** We implemented the hypervisor using BareMichael OS framework and run a Linux-based CNI service which has access to the network and file-system over the PCIe interface. The host emulates a boot interface.

our platform. Communication latency over the shared NoC is a potential new side-channel. Common countermeasures, such as those adding noise to VM timing information [27] or modifying VMs to inject noise on the usage patterns of the NoC [47], can be deployed to protect against such attacks.

5.4 Implementation

Our implementation setup assumes that the security enhancements described in Section 4.1 are available on the Intel SCC platform (i.e., we do not run the cloud architecture implementation on top of the emulated hardware changes).

Figure 9 illustrates the components of our implementation. The hypervisor is a single threaded bare-metal application based on the BareMichael framework [48]. This framework provides a simple operating system compatible with the individual cores of the Intel SCC. The hypervisor runs on core 0, the first core on the first tile. We load the hypervisor to the Intel SCC platform via the PCIe interface of the host. The hypervisor image also contains the code for the CNI service that, in our implementation, is an entire Linux OS (thanks to hypervisor factorization, the CNI service is not part of the TCB). At boot time the hypervisor sets up a logical partition for the CNI service. In particular, the hypervisor picks a free core (core 2) and allocates memory and network resources by configuring that core's LUT. Then, the hypervisor loads the CNI service image into the allocated memory and resets the core to start its execution.

After starting the CNI service, the hypervisor polls the MPB of the CNI core to check for VM launch requests. We implemented a small communication library for hypervisor–CNI interaction. This library differs from the standard Intel SCC communication model (and implementation) that assumes all cores can access each other's MPB which is not the case in our architecture. Instead, our library restricts the CNI service to accessing its own MPB for communicating with the hypervisor. This library uses the on-tile MPBs for data exchange and a shared lock for synchronization. By maintaining separate read and write buffers, the library enables full duplex communication.

The CNI service issues requests for allocating new cores, loading VM images, and starting VMs. The CNI service has access to a network-based file-system to fetch encrypted VM images. The hypervisor tracks the free cores and uses an example resource allocation policy that assigns to each core

	LOC
BareMichael framework	2557
LUT configuration	47
VM/CNI load	84
Core and MPB reset	35
CNI interface	123
Communication library	588
Total	**3.4K**

Table 1: **Hypervisor complexity.** We enhance the BareMichael framework with LUT configuration, VM load and core reset functionality. Additionally we implement a new communication library.

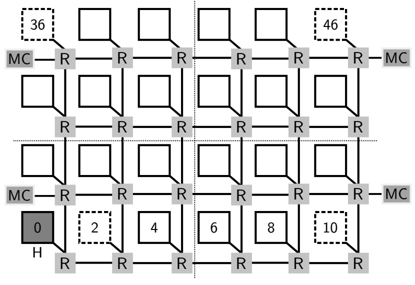


Figure 11: **Distance of cores.** We use cores 2, 10, 36 and 46 for our measurements. Core 2 is connected to the same memory controller as master core, while cores 10, 36 and 46 use different memory controllers.

a fixed-size memory region on its closest memory controller.

The lightweight enhancements we propose to the SCC platform enable logical partitions at the granularity of tiles (two cores). In our implementation we run a VM on the first core of the partition, while the second core is idle. In Section 6 we discuss operating systems that use multiple cores. Our implementation lacks a hardware cryptographic module and we implement all cryptographic operations using the PolarSSL library [6].

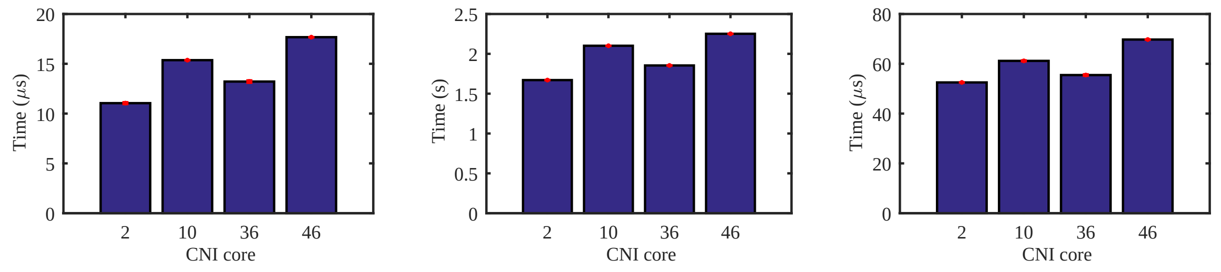
5.5 Evaluation

Hypervisor complexity. The BareMichael framework used as a basis for our hypervisor is 2557 LOC. We enhance it with LUT configuration, VM loading, and core reset functionality. We also implement a new communication library and CNI-interface. Table 1 shows the implementation size for each of these components. Our hypervisor is roughly 3.4K LOC in total, which is comparable to the smallest hypervisors available today [46].

Performance. We evaluate the time to setup the CNI service and the time to start a VM.

The time required to load the CNI service depends, naturally, on its size. We use a Linux OS that is roughly 32MB. The CNI setup time also depends on the location of the target core and its memory controller with respect to the master core (Figure 10). We set the master core to be core 0 and measure the CNI setup time for target cores 2, 10, 36 and 46 (as shown in Figure 11). Cores 10, 36 and 46 are the farthest from core 0 and also use different memory controllers.

Figure 10 shows the results of our CNI setup experiments. Each data point is an average over 100 measurements and



(a) LUT Setup (b) CNI Load (c) MPB Clear

Figure 10: **CNI service launch time.** Starting a CNI service on a core involves three main steps: LUT setup, image loading, and clearing the MPBs. The time required for each of these operations depends on the distance from the master core to the target core. Here we evaluate launch time to cores 2, 10, 36 and 46. We use a CNI service of 32MB and the overall startup time is between 1.7 s and 2.3 s

variances are negligible. It takes between 1.7 s and 2.3 s to setup the CNI service, depending on the distance between the master core and the target core. The CNI setup time is dominated by the time to load its image (Figure 10.b). We run the hypervisor on core 0 and we anticipate that running the hypervisor on a core in the center of the interconnect yields faster results.

Compared to the CNI setup time, starting a VM requires additional time. This overhead is due to the time required to decrypt the VM image and for the hypervisor and the CNI service to coordinate the launch of the VM.

As a VM image, we use an encrypted version of a Linux OS of 32MB. We load it from core 0 to core 46, the farthest core, which gives an upper limit of the loading time. Over 100 runs, it takes on average 43.47 s (± 5 ms) from the time when the CNI service requests to start a VM, to the time the hypervisor resets the target core of the VM. Most of the time is spent decrypting the VM image: 41.6 s (±4.6 ms). Since we emulate the cryptographic module, the decryption measurements are not representative of a real deployment scenario. On a hardware cryptographic module, similar operations would take less than a second [8] and the expected load time would be roughly three seconds.

Compared to the typical lifetime of a VM, we argue that even the overhead of our current implementation (with software decryption) is acceptable.

6. DISCUSSION

Partitions with multiple cores. Our design can accommodate logical partitions that encompass any number of cores, at the granularity of two cores. Running an OS on more than one core requires *(i)* that all the cores assigned to a partition have access to all the resources allocated to that partition, and *(ii)* an operating system that can run on multiple cores of an asymmetric processor. In the security-enhanced Intel SCC processor, the hypervisor can achieve the former by simply configuring the LUTs of all the cores within a partition to point to the same set of addresses. To realize the latter, one could use a distributed OS kernel like Barrelfish [10] or Helios [32].

Partitions for user code. We demonstrated how our security-enhanced Intel SCC processor can be used to run multiple operating systems in isolation in an IaaS cloud. However, current operating systems have grown complex and are prone to vulnerabilities [3, 5]. As a result, there is a need to create Trusted Execution Environments (TEEs) for running sensitive user code in isolation from a potentially untrusted operating system. In a cloud deployment, it is beneficial to create multiple such environments that scale with the number of VMs.

Many-core platforms are also available as co-processors, so they naturally provide an alternative execution environment to the host that runs the VMs. Furthermore, by carefully partitioning and adding cryptographic support to such co-processors, one could implement concurrent TEEs similar to Intel SGX [30]. The design of an architecture for isolating user-level code is beyond the scope of this work.

Processor vs. I/O virtualization. Our cloud architecture relies on I/O virtualization for enabling access to peripherals from VMs. The currently available virtualized peripherals replicate only the necessary subset of their functionality, and thus provide good scalability [16]. For example, the Intel Ethernet Controller XL710 supports up to 128 virtual instances. The goal of our work is to investigate similar resource minimization on the many-core processors. As we separate the functionality required for the hypervisor (master core) from those required for the VMs (other cores), the cores do not require additional execution mode or address translation. This is in contrast to current processor virtualization techniques, where both the hypervisor and the VMs run on every core. While a fully functional cloud architecture continues to require virtualization support on its peripherals, our approach enables simplification of the cores and thus improves overall system scalability.

Availability of user data. In this paper we focus on protecting the integrity and confidentiality of the user's VM. Ensuring availability is also an important factor in cloud deployments. In our architecture, providing data availability requires mechanisms such as data replication. Such mechanisms do not affect our architecture and can be seen as complementary to our work.

Resource utilization. While cloud architectures that leverage over-subscription can provide better efficiency, they rely on resource sharing that can lead to cross-VM attacks [44, 45]. The focus of this work is on cloud architectures where, thanks to assignment of dedicated resources and simplified resource management, several similar attacks are not possible and the system TCB size can be significantly reduced.

Future many-core platforms. Many-core processors are an emerging technology and there is uncertainty regarding optimal design choices for hardware and software on such platforms. A major issue for disagreement is the efficiency of synchronizing shared data structures across a large number of cores in hardware and software. For example, it is unclear if cache coherence would benefit or hurt the performance of processors with a very large number of cores [28, 29]. Therefore, it is hard to predict how future processors will be designed and how efficiently our solution can be ported on them. However, the main ideas underlying our design (NoC-based address space isolation and factoring of hypervisors into critical and non-critical services) are generic and likely applicable also to future many-core systems.

7. RELATED WORK

Centralized Hypervisors. Previous work on the design and implementation of centralized hypervisors such as NoHype [39], OSV [15] and MultiHype [36] are closely related to the solution that we present in this paper. NoHype relies on CPU and memory virtualization extensions and is based on the Xen hypervisor (above 100K LOC). OSV (8K LOC) uses a centralized hypervisor design to enable bare-metal execution of VMs, rather than isolation. It also relies on virtualization extensions during VM boot up and simply assumes that VMs do not misbehave at runtime. Finally, MultiHype relies on memory controller enhancements to realize logical partitions based on a centralized hypervisor design, but MultiHype is not disengaged (e.g., it manages I/O devices even if they are virtualization-capable) and its complexity is not publicly available. Unlike all these above proposals, our solution leverages simpler hardware support on an AMP processor (Intel SCC) to realize a small (3.4K LOC) and completely disengaged hypervisor.

Traditional Hypervisors. Traditional hypervisors for x86-systems leverage hardware virtualization extensions to achieve a small TCB. Examples include NOVA (20K LOC) [38] that uses a micro-kernel design to reduce the amount of privileged code and HypeBIOS (4K LOC) [46] that eliminates redundant code for the same purpose. Furthermore, formally verified kernels such as Muen [13] (complexity not publicly available) and seL4 [26] (8.7K LOC) as well as security-certified commercial hypervisors (e.g., Integrity Multivisor [2]) that rely on similar hardware virtualization extensions also exist. IBM and Hitachi today incorporate similar hypervisors into the firmware of their servers to achieve logical partitions [1, 19]. All of the above solutions are designed for SMP processors and it is unclear if similar designs would scale for use in future many-core systems.

Distributed Hypervisors. Similar to the architecture of future many-core processors, uncertainties exist regarding whether current hypervisor and more generally OS designs would scale for future processors. On the one hand, there is evidence that by changing existing operating systems, it is possible to enable their use on large-scale processors [12]. On the other hand, there is a trend towards pursuing alternate hypervisor and OS designs to achieve the same goal. This includes designing operating systems and hypervisors as distributed kernels whose components run on all or a subset of the cores [10, 32] and re-structuring them as services on separate cores that allow interaction via message passing rather than through traditional traps or exceptions that are handled on the same core [9,11]. An alternative approach for asymmetric processors is to run a set of distributed kernels that can provide shared virtual memory, i.e., a consistent view of memory across all cores like an SMP system [14,37].

All these distributed solutions aim to optimize the efficiency of the operating systems or the hypervisors, but not to achieve simplicity or disengagement. In fact, it is not possible to realize disengagement in a distributed solution, as discussed in Section 2.

8. CONCLUSION

We explored the problem of realizing logical partitions using small and disengaged hypervisors on many-core platforms. In contrast to current disengaged hypervisors that need full-fledged hardware virtualization extensions, our solution relies on a simple address-space isolation mechanism. We demonstrated this through a case study on the Intel SCC and showed that with lightweight modifications, it is possible realize logical partitions on many-core processors. We also designed and implemented a cloud architecture based on the security-enhanced Intel SCC. In this architecture, the hypervisor is completely disengaged, i.e., it is involved only in the startup and shutdown of VMs and the VMs run directly on the hardware without any virtualization overhead. The size of our prototype hypervisor is 3.4K LOC which is comparable to the smallest hypervisors known today.

Previously, many-core systems have been primarily used for parallel computing where efficient data sharing is needed. This work demonstrates that many-core processors can also be used in IaaS cloud deployments where secure isolation is a mandatory requirement.

9. ACKNOWLEDGEMENTS

We thank our shepherd Dr. Kurt Thomas for the very useful comments. We are also grateful to Dr. Devendra Rai and Dr. Stefan Lankes for access and help with the Intel SCC. This work was partially supported by the Zurich Information Security and Privacy Center. It represents the views of the authors. This work was also partly supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

10. REFERENCES

[1] Hitachi Embedded Virtualization Technology. http://www.hitachi-america.us/supportingdocs/forbus/ssg/pdfs/Hitachi_Datasheet_Virtage_3D_10-30-08.pdf.

[2] INTEGRITY Multivisor. http://www.ghs.com/products/rtos/integrity_virtualization.html.

[3] Linux Kernel Vulnerability Statistics. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[4] LPARBOX. http://lparbox.com/.

[5] Microsoft Windows 8 Vulnerability Statistics. http://www.cvedetails.com/product/22318/Microsoft-Windows-8.html?vendor_id=26.

[6] SSL Library PolarSSL. https://polarssl.org.

[7] Adapteva. Ephiphany Multicore IP. www.adapteva.com/products/epiphany-ip/epiphany-architecture-ip/.

[8] T. W. Arnold and L. P. Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 2004.

[9] Ballesteros, F. J., Evans, N., Forsyth, C., Guardiola, G., McKie, J., Minnich, R. and Soriano-Salvador, E. NIX: A

Case for a Manycore System for Cloud Computing. Technical report, Bell Labs Tech. J., 2012.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Symposium on Operating Systems Principles*, SOSP'09, 2009.

[11] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Operating Systems Design and Implementation*, OSDI'08, 2008.

[12] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Operating Systems Design and Implementation*, OSDI'10, 2010.

[13] R. Buerki and A.-K. Rueegsegger. Muen-an x86/64 separation kernel for high assurance. http://muen.codelabs.ch/muen-report.pdf, 2013.

[14] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-memory Multiprocessor. In *USENIX Annual Technical Conference*, USENIX'09, 2009.

[15] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu. A Lightweight VMM on Many Core for High Performance Computing. *SIGPLAN Not.*, 2013.

[16] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In *High Performance Computer Architecture*, HPCA '10, 2010.

[17] L. Fiorin, G. Palermo, S. Lukovic, V. Catalano, and C. Silvano. Secure Memory Accesses on Networks-on-Chip. *IEEE Transactions on Computers*, 2008.

[18] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. High-performance Hypervisor Architectures: Virtualization in HPC Systems. In *System-level Virtualization for High Performance Computing*, 2007.

[19] N. Harris, D. Barrick, I. Cai, P. G. Croes, A. Johndro, B. Klingelhoets, S. Mann, N. Perera, and R. Taylor. *LPAR Configuration and Management Working with IBM eServer iSeries Logical Partitions*. IBM Redbooks, 2002.

[20] Intel Corporation. Getting Xen working for Intel(R) Xeon Phi(tm) Coprocessor. https://software.intel.com/en-us/videos/knights-landing-the-next-manycore-architecture.

[21] Intel Corporation. Intel Xeon Phi 7100 Series Specification. http://ark.intel.com/products/75800/Intel-Xeon-Phi-Coprocessor-7120X-16GB-1_238-GHz-61-core.

[22] Intel Corporation. SCC External Architecture Specification. https://communities.intel.com/docs/DOC-5044/version.

[23] Intel Corporation. PCI-SIG SR-IOV Primer. http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf, 2011.

[24] Intel Corporation. Intel Virtualization Technology for Directed I/O. http://www.intel.com/content/www/us/en/embedded/technology/virtualization/vt-directed-io-spec.html, 2014.

[25] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A Fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*, SOSP '09, 2009.

[27] P. Li, D. Gao, and M. Reiter. Mitigating Access-driven Timing Channels in Clouds Using StopWatch. In *Dependable Systems and Networks (DSN)*, June 2013.

[28] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 2012.

[29] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 2010.

[30] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Hardware and Architectural Support for Security and Privacy*, HASP'13, 2013.

[31] A. Milenkoski, B. Payne, N. Antunes, M. Vieira, and S. Kounev. Experience Report: An Analysis of Hypercall Handler Vulnerabilities. In *Software Reliability Engineering (ISSRE)*, 2014.

[32] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Symposium on Operating Systems Principles*, SOSP '09, 2009.

[33] Partheymuller, Markus and Stecklina, Julian and Döbel, Björn. Fiasco.OC on the SCC. http://os.inf.tu-dresden.de/papers_ps/intelmarc2011-fiascoonscc.pdf, 2011.

[34] J. Porquet, A. Greiner, and C. Schwarz. NoC-MPU: A Secure Architecture for Flexible Co-hosting on Shared Memory MPSoCs. In *Design, Automation Test in Europe Conference Exhibition*, DATE'11, 2011.

[35] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Computer and Communications Security*, CCS'09, 2009.

[36] W. Shi, J. Lee, T. Suh, D. H. Woo, and X. Zhang. Architectural Support of Multiple Hypervisors over Single Platform for Enhancing Cloud Computing Security. In *Computing Frontiers*, CF '12, 2012.

[37] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A Case for Scaling Applications to Many-core with OS Clustering. In *European Conference on Computer Systems*, EuroSys '11, 2011.

[38] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *European Conference on Computer systems*, EuroSys '10, 2010.

[39] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Computer and Communications Security*, CCS'11, 2011.

[40] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. *SIGARCH Comput. Archit. News*, 2012.

[41] Tilera Corporation. TILE-GX Family. http://www.tilera.com/products/?ezchip=585&spage=618.

[42] TU Dresden. L4linux. http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[43] D. Wentzlaff and A. Agarwal. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 2009.

[44] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, Aug. 2014.

[45] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Computer and Communications Security*, CCS'12, 2012.

[46] Y. Zhang, W. Pan, Q. Wang, K. Bai, and M. Yu. HypeBIOS: Enforcing VM Isolation with Minimized and Decomposed Cloud TCB. http://www.people.vcu.edu/~myu/s-lab/my-publications.html, 2012.

[47] Y. Zhang and M. K. Reiter. Duppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Computer and Communications Security*, CCS'13, 2013.

[48] M. Ziwisky and D. Brylow. BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC. In *Many-core Applications Research Community Symposium*, MARC'12, 2012.