Isolated Execution in Many-core Architectures

Ramya Jayaram Masti
Institute of Information Security
ETH Zurich
rmasti@inf.ethz.ch

Devendra Rai
Computer Engineering and Networks Laboratory
ETH Zurich
raid@tik.ee.ethz.ch

Claudio Marforio
Institute of Information Security
ETH Zurich
maclaudi@inf.ethz.ch

Srdjan Capkun
Institute of Information Security
ETH Zurich
capkuns@inf.ethz.ch

Abstract

We explore how many-core platforms can be used to enhance the security of future systems and to support important security properties such as runtime isolation using a small Trusted Computing Base (TCB). We focus on the Intel Single-chip Cloud Computer (SCC) to show that such properties can be implemented in current systems. We design a system called SEMA which offers strong security properties while maintaining high performance and flexibility enabled by a small centralized security kernel. We further implement and evaluate the feasibility of our design. Currently, our prototype security kernel is able to execute applications in isolation and accommodate dynamic resource requests from them.

We show that, with minor modifications, many-core architectures can offer some unique security properties, not supported by existing single- and multi-core architectures, such as application context awareness. Context awareness, a new security property that we define and explore in this work, allows each application to discover, without any interaction with the security kernel, which other parts of the system are allowed to interact with it and access its resources. We also discuss how an application can use context awareness to defend itself from an unlikely, yet potentially compromised security kernel.

I. Introduction

A number of high-performance parallel computing platforms based on many-core architectures that are suitable for use in a wide range of applications have recently emerged. Examples of such systems include Adapteva's Epiphany co-processor [1] which is meant for use in cost- and power-efficient embedded applications and Intel's Single Chip Cloud Computer (SCC) [2] which is suitable for high-end cloud computing applications. These architectures consist of a large number of cores (*many* more compared to traditional multi-core systems) that communicate over a fast interconnect. Most such architectures are designed to heavily optimize inter-core communication using a number of techniques such as shared cache and message buffers, in order to improve the performance of parallel computing applications.

Given the potential for broad use of many-core architectures in a number of security and safety critical systems it is only natural to ask — Which security properties do these architectures offer and which properties could they offer? Examples of systems where many-core architectures are likely to be used include public clouds where Virtual Machines (VMs) are supplied by mutually untrusted parties, mobile platforms where installed applications have different levels of sensitivity and originate from sources with varying levels of trust and cyber-physical systems where applications have different criticality and corresponding priorities of execution [3], [4]. For simplicity, we henceforth use *applications* to refer to the software that executes in any of these three contexts. In all these systems, requirements for isolation between applications themselves and between applications and their underlying operating systems is one of the core security issues. It is clear that in order to limit the impact of the compromise of a subset of applications, it is desirable to provide each application with its own *securely isolated execution environment* which encompasses all the resources that the application requires.

In existing systems that use traditional multi-core architectures, as opposed to many-core ones, secure isolation is achieved by using a thin layer of trusted software often referred to as the system Trusted Computing Base (TCB). In order to manage and securely isolate system resources, the TCB runs at the highest privilege level and it is therefore desirable to keep it as small as possible, limiting the exposed attack surface. This is usually achieved by decomposing and restructuring traditional kernels (e.g. Linux) and retaining only security- and performance-critical functions within the TCB. Efforts in this direction have resulted in security kernels like seL4 [5], NOVA [6], and HypeBIOS [7] which are as small as a few thousand lines of code, orders of magnitude smaller than for example Linux. These existing solutions will now have to be ported by taking into account the nuances of many-core systems and their optimized resource sharing mechanisms. We show that many-core systems further simplify the design of existing TCB based solutions. This is because their hardware inherently supports efficient inter-process communication and features that can be leveraged to provide run-time isolation which are two important components of all existing TCBs.

Existing and upcoming multi-core architectures also provide hardware support for the creation of Trusted Execution Environments (TEEs) and therefore the isolation of applications; example technologies include Intel TXT [8], SGX [9], [10], [11], AMD SVM [12] and ARM TrustZone [13]. Intel TXT, AMD SVM and ARM TrustZone create TEEs by switching the entire platform between two states — one that is trusted to be secure and runs a thin TCB like the solutions presented above and another that runs an untrusted commodity OS. We show that many-core systems can provide similar guarantees at the granularity of individual cores instead of the entire platform, i.e., they can support many isolated execution environments that run in parallel. Furthermore, these isolated environments are more flexible than the ones that Intel SGX supports because they are not functionally restricted; for example, the secure isolated environments in many-core platforms can access virtualized peripherals and support secure input/output unlike Intel SGX.

In this work, we explore the security properties and opportunities enabled by many-core systems. We investigate design choices in the construction of many-core systems that allow secure but flexible partitioning of system resources between applications (or VMs, in a cloud-like scenario). Given the available cores and memory, we show that many-core systems can enable further simplification of existing TCBs thereby providing stronger isolation. Unlike existing TEE technologies, they can also be used to run large and demanding applications or VMs that span multiple cores in parallel and in complete isolation. We further show that many-core systems can not only support isolation, but also enable new security properties such as *application context awareness*. More specifically, we make the following contributions:

- By reviewing existing many-core architectures we show that they do not inherently provide strong security guarantees (e.g., per-core runtime isolation of applications). Memory and routing in these architectures has been designed and configured for high performance computing but not for efficient isolation of applications i.e., of cores and memory on which they run.
- We show that, with some modifications, many-core systems can be used to provide flexible and strong isolation of applications or entire systems that span multiple cores (e.g., VM instances or larger computing projects). We do this by designing SEMA, a system that modifies and extends the Intel SCC many-core platform. Our system consists of a modified Intel SCC platform which enforces isolation and a small security kernel that constitutes the software TCB and is responsible for system resource management. In contrast to existing TEE solutions, our solution enables multiple functionally unconstrained, isolated execution environments to run in parallel on their respective cores and allows them to interact with (virtualized) peripherals.
- We further show that since many-core architectures could be modified to support strong isolation of applications, they can reduce the complexity and thus the trust assumptions of the system's security kernel. Within SEMA we implement a minimal security kernel which sits on top of the modified Intel SCC platform. The functions of our kernel are limited to the assignment of applications to their respective cores and memory. We test the performance and evaluate the security guarantees that our system offers through a prototype implementation.
- Finally, we explore how many-core architectures can enable application isolation from the security kernel itself (to account for its possible compromise). We also discuss how to enable security properties like application context awareness, by which we mean the ability of a given application to become aware of which other applications share resources (e.g., memory) with it and which other applications have access to its state or communication. For example, a given application should be able to discover if any other application in the system or the security kernel has access to its memory. This is a useful property since it does not limit the ability of the applications to interact and share data and resources, but still enables them to have oversight over this interaction. We discuss which modifications of the Intel SCC platform are required to achieve context awareness.

Although different papers hinted at the possibility of using many-core systems for strong isolation([14], [15], [16]), to the best of our knowledge this is the first work that fully explores how this can be achieved and the security properties that many-core systems can provide.

The rest of the paper is organized as follows. We begin by presenting our problem statement in Section II. We then analyze commercially available many-core platforms and show that none of them can be securely partitioned into isolated execution environments using a centralized security kernel in Section III. In Section IV, we describe SEMA, an Intel SCC-based architecture that can create isolated execution environments using a centralized security kernel and analyze its security. We discuss extensions to SEMA that can be used to defend against a compromised security kernel itself in Section V. We present our prototype implementation of SEMA and evaluate its performance in Section VI. Finally, we describe related work and conclude in Sections VII and VIII respectively.

II. Problem Statement

Today's computing systems typically consist of a large number of software components that co-exist and share platform resources through an operating system. These components usually do not trust each other owing to different sources (e.g., applications in smartphones), different owners (e.g., user VMs in cloud computing environments) and different priorities (e.g., mixed-criticality applications in safety-critical systems). For simplicity, we refer to such software components in any of these contexts as *applications* henceforth. As already discussed in the introduction, one of the ways of limiting the damage that a

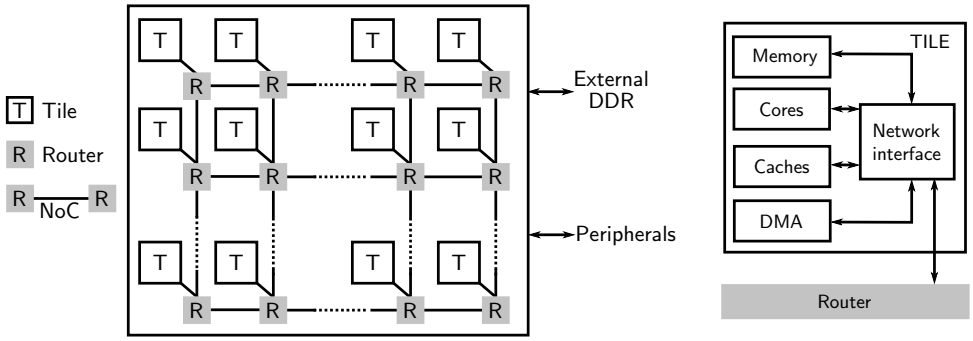


Fig. 1: A many-core processor is typically organized in tiles that are connected by a Network-on-Chip (NOC). Each tile contains one or more cores, caches, some local memory and a network interface that connects the tile to the NoC via the router.

compromised application can cause on a system is to provide each application with its own *isolated execution environment*. Such an environment should protects the application against external interference. On most computing platforms, isolation is managed by the system TCB consisting of operating systems or hypervisors, which make sure that applications are protected at runtime. However, a key limitation of this approach is the vulnerability of such operating systems and hypervisors themselves which results from their complexity. To address this problem, a number of technologies have been developed in the recent years to provide hardware support for isolation of a given application from other applications and from its underlying operating system or hypervisor.

Current processors include security extensions (e.g., Intel Trusted Execution Technology (TXT) [8], AMD Secure Virtual Machine (SVM) [12]) to bootstrap a trusted execution environment (TEE) on an untrusted operating system. These security extensions provide a dynamic root of trust for measurement (DRTM), i.e., they allow a user to verify the integrity of the target application at launch-time and protect it against run-time intervention from the OS or other co-resident applications. Recently, Intel announced a new range of processor extensions called Software Guard Extensions (SGX) that support multiple concurrent TEEs called enclaves [9], [10], [11]. Intel SGX protects the confidentiality and integrity of code and data within each enclave against unauthorized access from other enclaves and their underlying untrusted hypervisor. However, applications running within enclaves have no direct access to peripherals. In ARM TrustZone-enabled systems the processor supports two execution states [13], namely, secure world and normal world (or non-secure world). The processor switches between these states in a time-slicing manner, so that only one state is active at a time. The normal world runs the OS and regular applications on top of it, whereas the secure world runs trusted applications. The latter run on top of a small layer of software called the trusted OS. Software running in the secure world is isolated from anything running in the normal world. Existing multi-core systems further allow the assignment of cores and memory to for example VMs using a disengaged hypervisor [14]. All of these technologies and approaches consider only single and multi-core systems. They neither consider how to achieve isolation on many-core architectures nor additional security properties that many-core architectures could enable.

In this paper, we fill this gap and explore how many-core systems can enable several security properties. As we already described, the support for runtime application isolation has already been explored in the context of single- and multi-core systems. The first problem that we therefore want to address is how isolation using a small security kernel can be efficiently achieved within many-core systems. The second problem that we want to address is the one of application isolation from a possibly compromised security kernel. To solve this problem we explore how to restrict kernel capabilities such that it cannot harm the applications it manages, and how to enable applications to be aware of the context within which they execute. We note that context awareness was, so far, not considered in any architecture and emerges as a new property that can be enabled by many-core systems.

In what follows, we describe the properties that are desirable in any isolation solution for many-core architectures.

Flexibility: The architecture should allow the creation of flexible and dynamic execution environments, i.e., execution environments with varying resources (cores, memory, access to peripherals) on demand. Each execution environment should be securely isolated from others and must incur *negligible* overhead due to the isolation enforcement mechanisms.

Small TCB: Since the integrity of the security kernel which constitutes the system TCB is vital to the security of the platform, it is necessary to keep it as small and low in complexity as possible. It is also necessary to minimize its interaction with any other co-resident (and potentially malicious) software so as to minimize the risk of its compromise. This can be achieved by using the kernel only for scheduling and resource management in the system and by allowing applications to execute independently without run-time kernel support, except when they need to request additional resources.

Restricted Kernel Capabilities: It is often necessary to minimize the impact of kernel compromise in order to limit the damage to the applications it manages. Ideally, a compromised kernel should be able, at most, to stop the execution of a given task or decide to not schedule it at all. It should never be able to interfere with an application's execution, access or modify the

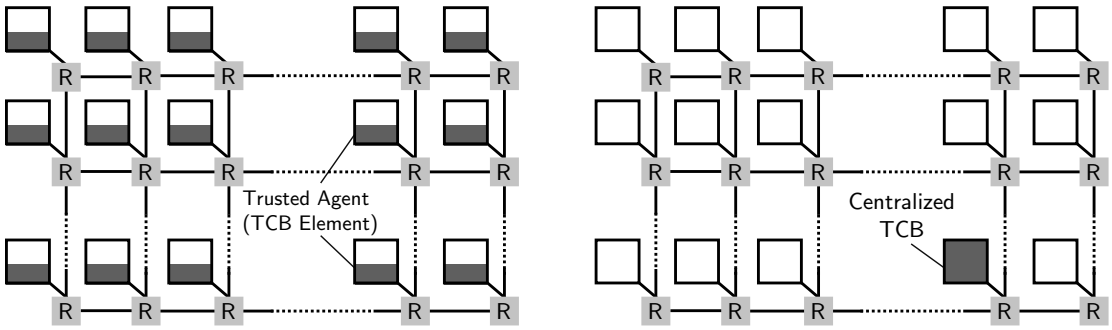


Fig. 2: Many-core systems can be partitioned into isolated execution environments either by using a distributed software TCB in which a trusted agent runs on every core (left) or a centralized TCB that runs in its own environment and leverages hardware support to enforce run-time isolation between other environments (right). Here we assume that each tile has a single core.

application's state. One way to achieve this is to restrict the capabilities of the kernel itself so that an attacker who gains control over it has limited options and cannot for example access the application's memory.

Context Awareness: The architecture should provide each application with a mechanism to detect all other applications that have been configured to interact with it (e.g., share memory with it). It would be up to the application currently running, to decide how to act upon discovering that a new application is able to interact with it. Currently, no architecture allows for context awareness without the direct help of the highest-privilege software, i.e., the kernel running in the system. Therefore, once the kernel is compromised, an application would not be able to know which other applications are able to access its memory and other resources. One benefit of application context awareness within many-core systems is that it can provide each application with ways to detect if other applications or the kernel are trying to violate the confidentiality and integrity of its data and code but without limiting the ways in which applications can interact. An application can then react appropriately (e.g., skip sensitive operations) if it discovers suspicious interactions/accesses.

III. Analysis of Existing Many-Core Systems

In this section, we review existing many-core architectures and analyze the feasibility of achieving the security properties discussed in Section II on them.

A. Solution Space

Most many-core processors follow a *tiled* architecture as shown in Figure 1. Each tile consists of one or more cores and a network interface that connects the tile to the on-chip network. Each tile may additionally contain one or more caches (either integrated into each core or shared by all cores on the tile). It may also include a small on-tile memory that is either partitioned between or shared by all cores on the tile; this memory is typically used for message passing between cores in multi-processing applications. Alternatively, tiles communicate via a shared on-chip memory. Furthermore, all cores have access to one or more large off-chip (DDR) memories which are used to store application code and data. Finally, each tile may optionally include a Direct Memory Access (DMA) engine used for efficient data transfers either between tiles or between the tiles and off-chip components (e.g., memory, peripherals). DMA engines allow for separation of communication and computation tasks and are, therefore, useful for efficient multi-processing.

The network interface at every tile connects it to the other on-chip (e.g., other cores, shared on-chip memory, etc.) and off-chip (e.g., DDR memory) components via the Network-on-Chip (NoC). This network consists of a set of connected routers (one per network interface) that are responsible for transferring data among tiles and between the tiles and off-chip components. Each router typically contains five interfaces (North, South, East, West, parent-tile), uses a static routing algorithm and a round-robin scheduling algorithm to transfer packets arriving at its different interfaces. The NoC is a high-bandwidth network and allows efficient transfer of large data with very low latency.

Access to off-chip components is realized by forwarding the relevant traffic to the routers at the periphery of the chip which connect to those components. Examples components include memory controllers that mediate access to DDR memory, network interfaces, etc.

The space of solutions that achieve secure isolation on many-core systems can be roughly divided in two categories: *(i)* solutions that require a trusted agent/TCB element on every tile and *(ii)* solutions that only require a centralized TCB that executes in its own isolated environment. We illustrate these solutions in Figure 2. Given the security properties that we want to achieve, the design and potential use of many-core architectures, especially cloud computing, in this work we opt for solutions that rely on a centralized TCB. Such solutions, compared to those that rely on a distributed TCB, will *(i)* result in a smaller TCB, *(ii)* reduce the interaction between the applications and the TCB or allow better TCB disengagement, *(iii)* allow

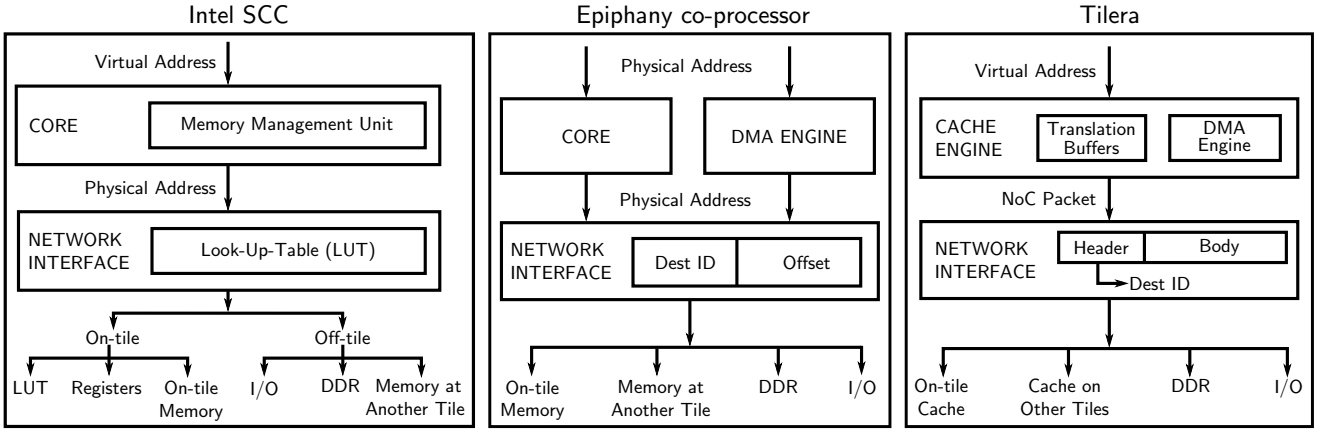


Fig. 3: Address space isolation is key to achieving secure isolation on many-core architectures. The process of address translation in currently available commercial systems is freely configurable by any software that is running on any core of the system (like in Intel SCC that uses Look-Up-Tables), completely static (as in the Epiphany co-processor) or controlled by a privileged entity on each tile like in Tilera's Tilepro processors. None of them currently support address space isolation using a centralized TCB.

us to address the privacy concerns of individual execution environments (e.g., in the cloud) and finally, *(iv)* help in limiting the impact of a malicious TCB.

However, realizing centralized solutions requires support from the underlying hardware to securely isolate individual execution environments. In particular, the hardware needs effective mechanisms to control data flow and access within and across different tiles — this is especially important in the case of many-core systems which feature hardware support for data sharing among tiles for efficient multi-processing. Below, we look at commercially available systems and analyze if they provide support for the creation of multiple isolated execution environments using a centralized TCB.

We focus on three popular systems: Intel's Single-chip Cloud Computer (SCC), Adapteva's Epiphany chip and Tilera's line of many-core processors. We chose these processors because they are commercially available today and have contrasting designs — these systems vary in terms of the number of cores per-tile, cache architectures, memory access and sharing mechanisms.

B. Analysis of Commercial Systems

Most many-core architectures use memory-mapping for access to all system resources. This includes access to the on-tile resources like local memory and registers as well as off-tile and off-chip resources like the memory on other tiles, DDR memory and I/O peripherals. Therefore, isolating different execution environments essentially reduces to isolating address spaces. In order to determine the feasibility of creating isolated execution environments on commercial many-core processors, we look at how they create and manage different address spaces in the system (Figure 3). Note that this address space partitioning is typically the responsibility of the security kernel (also called the software TCB), which is a small but privileged layer of software that configures and controls the system. We summarize our findings below.

Intel Single Chip Cloud Computer (SCC): Intel SCC [2] is a co-processor that consists of 48 x86 cores that are organized in 24 tiles (two per tile). Each core has a memory management unit that translates virtual addresses to physical addresses. These physical addresses in turn get translated into a system-wide address using a Look-Up-Table (LUT) at the network interface. Each core has its own LUT and is able to read as well as modify it from software (both from user and supervisor mode) at run-time. Therefore, each core can access *any* system resource except the caches on each tile by simply modifying its LUT. Each LUT entry points to a 16MB region of memory. We note that the LUTs of both cores on a tile are memory mapped to the same 16MB region.

Interestingly, access to the LUT itself is configured through an entry in it and therefore, it is possible to prevent a core from accessing its own LUT by simply removing this entry. However, this also deprives the core of access to important registers (for interrupt, power, frequency management) that reside in the same memory region as the LUT itself. If software on the core requires access to these registers, then granting it access to only the registers (and not the LUT) requires a trusted layer of software that intercepts any access to the LUT. In other words, it requires a security kernel element on each tile to enforce isolation.

Furthermore, the LUTs in Intel SCC can also be used to enable application context awareness — an application on a certain core can read the LUTs of all other cores in the system to determine if it shares resources with any of them. However, this also allows it to learn about all the resources that other cores have access to and even modify them. Such unlimited access to LUTs of other cores is undesirable from the security point of view. Finally, Intel SCC does not also have mechanisms to limit the privilege of the security kernel itself.

Epiphany Co-processor: Adapteva's Epiphany co-processor [1] contains between 16 and 1024 RISC cores (similar to ARM) and is targeted at low-power embedded systems. The cores are 32-bit and distributed into separate tiles. Each core only uses a flat 32-bit address space that is partitioned between local on-tile memory, memory on other tiles, external DDR and peripherals. The network interface uses an implicit and fixed set of rules to determine the target, i.e., it uses the top 12 bits of every address to find the destination tile. Since the network interfaces are not programmable, the default settings that allow every core to access *all* system resources cannot be changed. This implies that one would have to run a privileged kernel component at every core to restrict access to system resources.

Furthermore, the Epiphany co-processor allows DMA transfers both from and to the local memory on each tile. Since such DMA access bypasses any kernel component running on the core, it may not be possible to achieve isolation even with a software TCB element on each core but this remains to be verified[1]. Lastly, the Epiphany co-processor contains no features that can be used to enable application context awareness or restricting the capabilities of a potentially compromised security kernel.

Tilera's TILEPro Processors: Tilera's many-core processors [17] contain up to 64 tiles with one core each. Each core follows a Very Long Instruction Word (VLIW) architecture and has its own cache and on-tile routing engine. Each core also supports memory virtualization through address Translation Buffers (TLBs). These TLBs can only be configured by software that runs in the second highest privilege mode of the processor called the hypervisor mode (the highest mode is used for debugging the hypervisor). Therefore, by running a software kernel agent that runs in hypervisor mode and using it to configure the TLBs, it is possible to limit a core's access to system resources. The hypervisor can also configure the DMA-engine on each tile to support IOMMU-like features. However, this does not satisfy the requirement of a centralized TCB.

We note that Tilera's processors support mechanisms to disable communication between adjacent tiles in hardware. Together, with Tilera's hypervisor that runs on every core, these hardware features can be used to create isolated execution environments [18]. However, the possible layout of such environments is restricted in the sense that they have to be contiguous. Finally, the Tilepro processors have no mechanisms that can be leveraged to achieve application context awareness without the help of the security kernel or to contain the privileges of such a hypervisor or any other security kernel in the system.

In summary, all the above commercially available many-core processors require a kernel component running at each core in order to create isolated execution environments. Furthermore, none of these systems inherently support context awareness for applications although they (at least Intel SCC) could be easily modified to achieve it. Finally, these current systems do not provide any means to limit the damage resulting from a compromised security kernel.

Although centralized and distributed TCBs have their own advantages and disadvantages, given the space of applications that we target in this work (i.e., cloud computing), we focus on designs based on a centralized TCB. A thorough comparison in terms of security, flexibility and performance between the two approaches is intended as part of future work.

IV. SEMA: A Secure Many-core System Architecture

In this section, we present SEMA, a system that extends Intel SCC and supports flexible creation and management of isolated execution environments on this many-core architecture. SEMA relies on a centralized security kernel that is in charge of creating, managing and tearing down execution environments. We designed the system such that the kernel runs on one of the system tiles and the applications run on other cores and resources assigned to them by the kernel. We assume that all the peripherals connected to the SCC are virtualized, i.e., that they expose a dedicated interface to each client (core in this case) and allow concurrent requests from multiple clients without the engagement from the TCB. As we already noted, such a centralized solution fulfills well the requirements imposed by for example cloud computing systems where secure isolation between VM instances as well as their isolation from the TCB is strongly preferred. The choice to implement SEMA as an extension to Intel SCC was natural since this architecture required minimal changes to achieve the desired security properties. In what follows, we describe the Intel SCC architecture in detail and highlight our extensions to it.

Figure 4 presents a schematic of SEMA. Underlying our system is an Intel SCC, composed of 24 tiles organized in a 6x4 matrix. Each tile contains two cores, each with their own L1 and L2 caches. Each tile also has some local on-tile memory that is used by cores for efficient message-passing between tiles and is called the Messaging Passing Buffer (MPB). The network interface at each tile, also called the mesh interface, contains one Look-Up-Table (LUT) per core that is used to translate the physical address issued by a core to a *system* address. A system address could point to the MPB, memory-mapped registers on the tile, the LUT itself, DDR memory or memory-mapped peripherals. Each LUT entry points to a 16MB region of memory — in other words, this is the granularity of memory access control.

Intel SCC has four on-chip memory controllers that serve as gateways to four different DDR blocks each up to 8GB in size. By default, the entire system is divided into four quadrants each corresponding to one memory controller. All cores in a quadrant are mapped to the same memory controller by default. However, the mapping of cores to memory controllers can be modified by changing their LUTs.

Intel SCC's LUT-based design is very conducive for flexible and efficient system partitioning. Since access to all system resources can be controlled through the LUTs, it is practical to implement security policies in them. However, currently LUTs

[1]We have not yet been able to obtain a prototype board for testing.

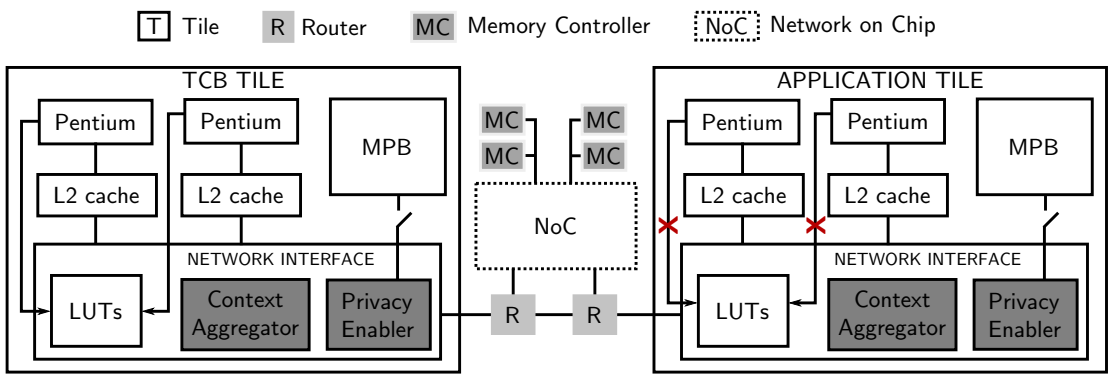


Fig. 4: In SEMA, the security kernel runs on a single tile and is responsible for setting up and dispatching applications to other cores. Only the kernel is allowed to modify the LUTs in the system and therefore, any dynamic modification to them has to be mediated by it. However, to defend against a compromised security kernel, each core in the system would need to support TEE-like features. This combined with our extensions to the network router (shown in dark grey on the right) at each tile help achieve run-time isolation even in the context of a compromised kernel by enabling context awareness for applications that run on the cores and limiting access from the kernel to each core's local memory.

are writable by the cores themselves and preventing this results in loss of other functionality (for details, see Section III). To overcome this limitation, in SEMA we assign the LUT of each core to a dedicated 16MB block to which that core can be denied access or given read-only permissions. Furthermore, unlike the current design, the LUTs of the two cores on the same tile are assigned to different 16MB blocks to prevent unwanted interference between cores. The same reasoning applies to the MPBs on every tile. Any attempt to access resources that are not reachable via the LUTs essentially results in the termination of the application that violated the security policy. This run-time isolation is enforced by the LUTs themselves during address translation as shown in Figure 3.

The security kernel in SEMA performs three major functions: *(i)* scheduling *(ii)* resource allocation and *(iii)* security (LUT) configuration. Note that in this work, we do not focus on the scheduling algorithms that the kernel chooses to implement. The kernel first decides when and on which core(s) an application will execute, allocates resources to it, configures the hardware to enforce run-time isolation and finally, allows the application to execute. It does not offer any run-time services to the application other than dynamic resource allocation. The kernel itself occupies a single tile and is the first to execute after system boot.

A. SEMA: Boot and Operation

At boot, in SEMA, the security kernel is loaded into the DDR and it begins execution on one of the tiles in the system. When it schedules a new application for execution, it allocates one or more cores, dedicated DDR memory and instances of virtual devices to the application. It then configures the LUTs of these allocated cores to ensure that they all have access to the each other's resources (MPB, DDR, etc.). The kernel then loads the application into the assigned memory and resets the corresponding cores to start its execution. These are the only steps required to run any software ranging from a simple bare-metal program to a full Linux OS on a new core. The application now executes with the resources allocated to it until it terminates voluntarily or is stopped by the kernel. Note that the kernel can terminate applications on other cores and halt them by raising a non-maskable interrupt (NMI).

The application that is started by the kernel can independently administer all the resources allocated to it. However, it cannot access or modify any resource to which it was not given access through its LUT configuration. Any attempt to do this results in the application being terminated. Thus, multiple applications can execute in their own independent, isolated environments concurrently without interfering with each other. The same guarantees apply to the protection of the kernel itself against the malicious applications that it may schedule and run.

Finally, in SEMA, an application running in its own execution environment can request more resources from the kernel. To accommodate these requests, the kernel uses the same procedure described above to allocate additional resources. The simplicity of partitioning system resources is a result of the LUT-based architecture that represents a central access point to *all* system resources. This allows flexible and dynamic sharing as well as firewalling of resources. It also enables the creation of isolated execution environments that can scale dynamically in a secure manner.

B. Security Analysis

In this section, we briefly analyze SEMA and show that it achieves some of the security properties discussed in Section II. We begin by presenting our assumptions about the attacker's capabilities.

We consider an adversary who controls, or by remote compromise gains control, over one or more applications which execute alongside security-sensitive applications, all running on the SEMA system. The goal of the attacker is to obtain access to or modify any sensitive information in the sensitive applications or interfere with their execution. We however assume that the adversary does not have physical access to the SEMA hardware and therefore cannot compromise it nor launch any physical attacks against it. Finally, we assume that the hardware itself is trusted and functions as expected.

In order to interfere with a sensitive application, the adversary must be able to gain access to one or more of its resources. Given that in SEMA, access to all resources is configured using the LUTs, the adversary should only be able to modify the LUTs of one or more cores that he controls. However, in SEMA, only the system's security kernel has access to the LUTs of all cores in the system and therefore, only it can make modifications to them. An attacker cannot gain access to the resources of any other application and cannot interfere with it. This protection equally applies to the kernel itself. Moreover, since the security kernel can create environments of varying resource configurations by appropriately configuring the LUTs, SEMA satisfies the property of *Flexibility*. By design, SEMA disengages the security kernel which only does scheduling and resource allocation in the system and therefore has a *Small TCB*.

As presented so far, SEMA does not provide any protection against the compromised security kernel. In the next section we show how our system can be extended to provide this protection as well and satisfy the remaining properties listed in Section II.

V. Defending Against A Compromised Security Kernel

SEMA uses a trusted security kernel to create and isolate execution environments from each other and relies on the minimal and disengaged nature of the kernel to reduce the risk of its compromise. Nevertheless, it is desirable to have an architecture that can limit the damage that can be caused by an attacker who gains control over the kernel. More specifically, a compromised kernel should at most be able to stop a task or not schedule it at all — there is little we can do to prevent this because the kernel is responsible for resource allocation and scheduling. However, the compromised kernel should not be able to access or modify sensitive information belonging to the application or influence the application's execution, i.e., we must at least be able to still achieve isolation guarantees in the event of kernel compromise. Furthermore, an application should also be able to detect a misbehaving kernel by leveraging on the context awareness feature supported by the underlying hardware. Below, we discuss how these two properties can be achieved in the context of Intel SCC.

Isolation: In order to achieve isolation, an application has to protect all the resources — both on-tile and off-tile components in its execution environment. As a first step, we focus only on achieving isolation of application's on-tile resources, namely, its core, caches and MPB.

Since the kernel is compromised, it could inject (highly privileged) malicious code into any application's execution environment. Therefore, we require a mechanism that is similar to TEE construction using Intel TXT [8], Intel SGX [9], [10], [11] or CARMA [19] that allow us to construct a TEE by circumventing a malicious OS/hypervisor on traditional multi-core systems. In fact, CARMA was designed to offer TEEs that are limited to the on-die components which is similar to creating a TEE on a tile. It essentially uses the Cache-as-RAM capabilities that is available in modern CPUs to create a TEE. Such a TEE would also require support from the network interface to block all accesses to on-tile resources from outside (say using a privacy enabler as shown in Figure 4).

Context Awareness: Context awareness essentially allows an application to learn about all the resources that it shares with other execution environments in the system. This allows the application to e.g., detect that a misbehaving kernel which misconfigured resources that were not meant to be shared.

Since all the information about access to different system resources is contained in the LUTs, aggregating this information allows a core to determine with which other cores it shares these resources. However, it is also undesirable for a given core to learn about the LUT entries and in turn, all the resources of other cores in the system. Therefore, we propose extending the network interface on the tile with a hardware-based *context aggregator* that provides a special service — it gathers the LUT information of all other cores through its router, filters out all entries that do not have a match in the requesting core's LUT and passes this information to that core. As a result, a core only learns *if* it shares resources with any other core but nothing more. A core could access this functionality by using the TEE environment that can be created using features like CARMA as we discuss above. Otherwise, a malicious kernel can intercept the messages from the network interface and modify them before they reach the application that runs on the core.

We note that this approach could lead to Time-Of-Check Time-Of-Use (TOCTOU) attacks, i.e., the compromised kernel could modify the LUTs just before the check is performed. However, this would require that the kernel learns about when this check occurs. One way for the kernel to do this is to monitor the traffic between the context-aggregators at different routers and this can be prevented by ensuring that the routers do not forward any information not destined for their own tile to the software on that tile, i.e., preventing routers from working in promiscuous mode. The only other way for the kernel to learn about the check is to be running on the target application's tile but this can be prevented by using TEE techniques like CARMA or Intel TXT (on one core).

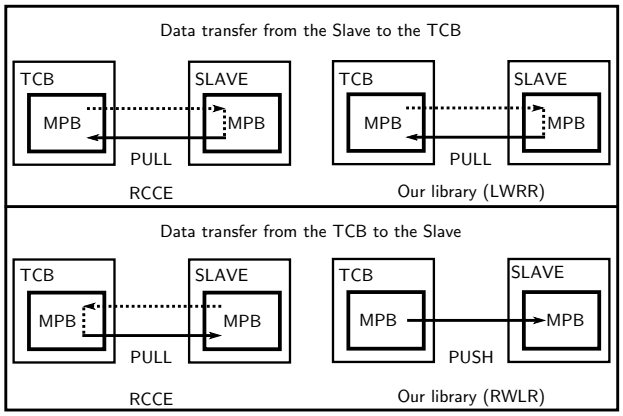


Fig. 5: Currently, Intel's RCCE library for communication between cores assumes that all cores in the system can read and write each other's MPBs. However, this is not desirable for communication between the kernel and any slave application because it requires the latter to have read-write access to the kernel's MPB. Our new communication library overcomes this drawback and restricts slave cores to using local reads and writing only to their MPBs.

VI. Implementation and Evaluation

Although the changes required to Intel SCC to realize the SEMA architecture are quite small, they still require changes to the SCC hardware. Given that Intel SCC is an experimental platform, we believe that the changes to it that we propose are small enough to be incorporated into a future product. Here, we prototype and evaluate important components of the SEMA architecture assuming that these hardware changes are in place. We adopt this approach for two reasons: first, the components that we implement will not change much even when incorporated into a full prototype and second, the system performance is not affected by the modifications that we propose.

A. Our Prototype

Our prototype consists of two main components: a simplified kernel and a secure communication library. We describe them in detail below.

Simplified TCB: Our security kernel is a single threaded bare-metal application based on the BareMichael framework [20]. The kernel runs on core 0 — the first core on the first tile of the Intel SCC. Since Intel SCC was designed as a co-processor, it does not have a BIOS of its own and also lacks access to disk (the abundant RAM is used as a disk dump). Applications can only be loaded for execution on the SCC via the PCI Express (PCIe) interface that connects it to the host. We, therefore, load our simplified kernel using this interface. We note that we only have remote access to the Intel SCC board via Intel's University program and hence cannot modify this setup in any way.

The security kernel can be configured to load either a full OS (like Linux) or a bare-metal application on to a slave core. For brevity, we refer to software that is loaded by the kernel as a slave application. The kernel first sets up the slave core's LUT, loads the slave application into the target core's memory and resets the target core to start execution. Additionally, the kernel allows a slave application to request more resources (cores). The kernel also assists the slave application in the setup and configuration of its newly granted resources. More specifically, it sets up the LUTs of the extra cores granted to the slave application on its behalf. Note that the slave application by itself cannot do this because by design, SEMA allows only the kernel to write LUTs in the system. The kernel also configures sharing of MPBs or DDR memory between the slave application and its newly allocated cores.

Secure Communication Library: We implement a secure communication library that allows slave applications to interface with the security kernel. Such communication may be necessary, for example, when requesting more resources, sharing existing resources with other applications, etc.

Intel's original communication library for SCC is called RCCE and it enables communication between cores of the system through the MPBs. RCCE assumes that all cores in the system have access to each others MPBs and can read as well as write to them. As shown in Figure 5, RCCE uses a local-write, remote-read policy; i.e., a sender writes his data locally and notifies the receiver who then *pulls* the actual data. This is not suitable for communication between the kernel and its slave applications because it entails allowing the slave applications to read and write the kernel's message passing buffer.

To overcome this problem, we implemented a new communication library that requires any slave core to only read from and write to its local MPB (Figure 5). The security kernel in turn polls each slave's MPB for a new message and reads available messages from it directly. Similarly, it also writes messages meant for a slave directly to the slave's MPB. In other words, unlike

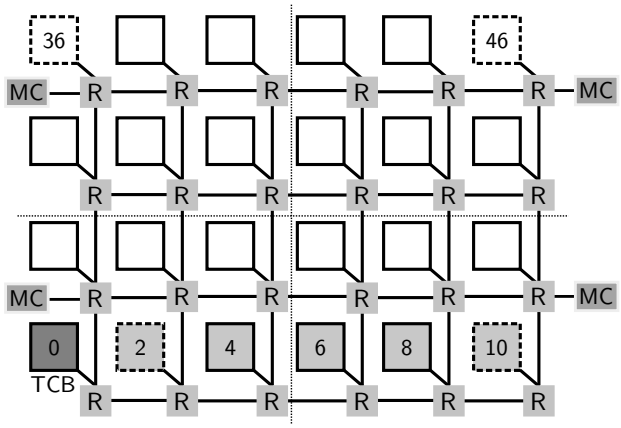


Fig. 6: We evaluate two aspects of the performance of our prototype: first, the slave application setup time (i.e., the time required to load an application on to another core and execute it) and second, the inter-core communication latency when our library is used. We measure these values for different distances (different number of hops) between the security kernel and the slave core (and its DDR memory). We choose one core on each tile outlined by dotted lines for the first test and a core on the tiles filled in grey for the second test.

RCCE that only allows data *pulls*, our library allows data *pulls* which we refer to as Local-Write-Remote-Read (LWRR) and data *pushes* which we call Remote-Write-Local-Read (RWLR). Payloads are transferred in chunks of 64 bytes as this is the maximum buffer size available per core on partitioning the MPB (4KB) into 96 parts (one read and write buffer for each core on the Intel SCC). Note that by maintaining separate read and write buffers per core, the library enables full duplex communication.

B. Code Complexity

Since code size is often used as an indicator of code complexity which in turn is directly related to the risk of compromise, here we discuss the size of the security kernel and the communication library in our prototype.

TCB Complexity: Although we did not implement a complete security kernel which constitutes the software TCB, our prototype allows us to measure the complexity (in terms of Lines-Of-Code/LOC) of the process of setting up the execution of a slave application. This essentially involves three functions: *(i)* LUT configuration of slave cores, *(ii)* slave application/OS load and *(iii)* resetting the core(s) assigned to the application. The *memcpy* code module[2] which is used by these functions is around 50 LOC.

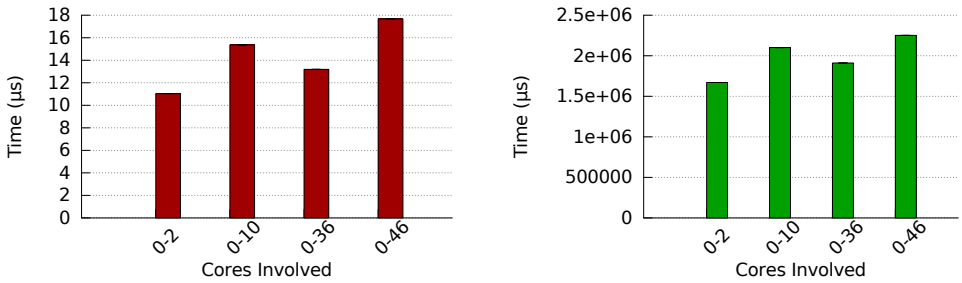
The LUT of each core has 256 entries each of which is one word (32-bit) in size. The complexity of the LUT configuration function depends on the number of LUTs written. In our system, we use a filtered version of Intel's default LUT configuration that maps the LUTs and MPBs of all tiles into each core's LUTs. More specifically, we map about forty entries that together enable access to 640MB of DDR, two entries that point to the LUTs itself, two entries for the MPB, 5 entries to the different system interfaces (network card, host controller, etc.) and one LUT that points to the boot segment when we set up the LUTs of a core. This takes about 46 LOC per core.

The complexity of the code used to load the slave application depends upon its size. The actual load process involves two steps: first, the kernel remaps one of its own LUT entries to the target DDR memory and then copies over code to that segment. The kernel repeats this for every code segment in the slave application. Our prototype uses about 50 LOC to map one segment of memory. Resetting a core involves only flipping a bit in one of the core's registers after computing its address and takes about 20 LOC.

Finally, our prototype security kernel also contains code to communicate with its slave applications and grant them extra resources. Its overall size (allowing for debug output) is about 700 LOC. As we have described we make use of the BareMichael bare-metal framework to execute our kernel. We did not change its code-base which contains around 2693 LOC. We note that our code can be further optimized to reduce the size of the kernel even further.

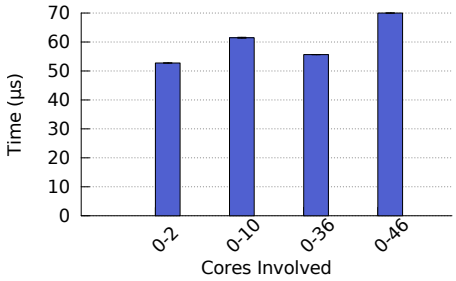
Complexity of the Secure Communication Library: The secure communication library contains five main functions. The first function is used to setup the flags and buffers required for communication in the MPB as part of the library's initialization. The library also contains two functions for reading and writing remote MPBs which is used by the kernel and two other functions for reading and writing local MPBs which is used by the slave applications to communicate with the kernel. Finally, the library also includes Intel's *memcpy* implementation. Together, they constitute around 500 LOC.

[2]We used Intel's optimized memcpy that we adapted from the RCCE library in our implementation.

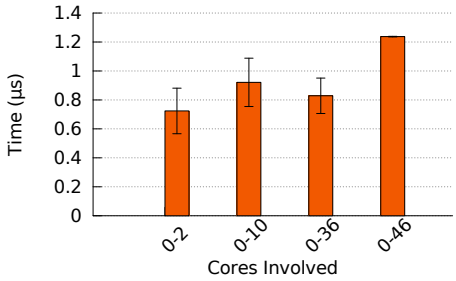


(a) LUT Setup

(b) Linux Load



(c) MPB Clear



(d) Reset Core

Fig. 7: Loading the Linux OS from the security kernel for execution on a slave core (in this case, cores 2, 10, 36, 46) involves four steps: LUT setup, loading of the linux image into DDR, clearing the MPBs of the slave core and finally resetting it. The time required for each of these functions depends on the distance (hop count) of the slave core and its memory controller from the kernel.

C. Performance Evaluation

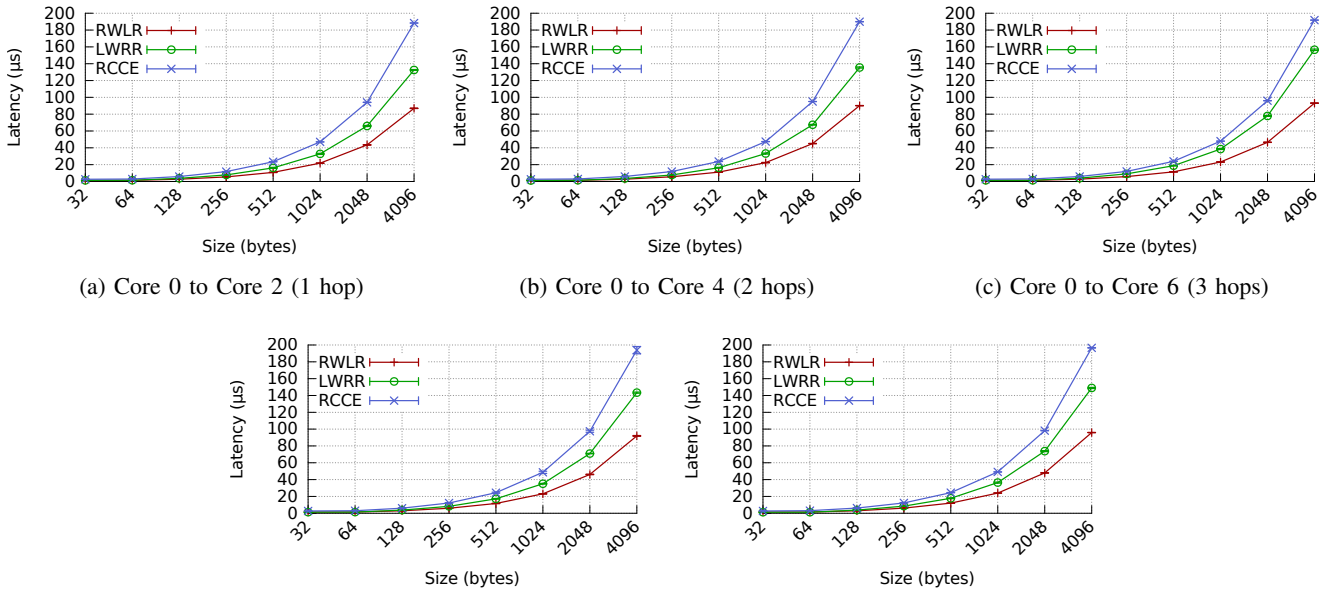
We evaluated the performance of our prototype with respect to two metrics, namely, the slave application setup time and the data transfer latency between cores using the secure communication library. We discuss the results of our experiments in detail below.

Slave Application Setup Time: The slave application setup time is the time required to dispatch and begin execution of the application on one core. In our experiments, we measure the time it takes to load a Linux OS which is roughly 32MB in size and is divided into 6 segments. The time required to load Linux depends on two main factors: the location of the core and the DDR memory (or memory controller) that the core uses. So we choose the experimental setup shown in Figure 6 as it explores the corner cases with respect to both these factors. More specifically, we load Linux from the kernel running on core 0 to cores 2, 10, 36 and 48. Cores 10, 36 and 46 are not only farthest from core 0 in terms of hop-count (in the x- and y-directions and in total, respectively) but also use different memory controllers compared to core 2 which is only 1-hop away from the kernel and also shares its memory controller.

Figure 7 shows the results of our experiments. Each data point in these plots is an average over 25 measurements. We note that the variance in most of the values is very small and not visible in the plots. Clearly, the LUT configuration, MPB initialization and core reset depend upon the location of the target core. As its distance from the kernel's core (in terms of the number of hops) increases, the time required for these tasks also increases. Note that the setup time for core 36 is smaller than core 10 because it is only 3 (along the y-axis) hops away as opposed to core 10 which is 5 hops (along the x-axis) away from the kernel on core 0. The application load time depends on the distance between the memory controller of the target core's DDR and the kernel. As the hop count between the target memory controller and the kernel increases, the application load time also increases.

In summary, it takes between 1.5 and 3 seconds to load a Linux OS onto a core in the platform from the kernel running on core 0. We postulate that running the kernel on a more central core (say in the 2nd or 3rd row and on the 3rd or 4th column) would result in more efficient loads because it requires a reduced number of hops to reach every other core on the system.

Inter-core Communication Latency: The data transfer latency between cores using the new communication library is the amount of time it takes to transfer payloads of different sizes between the MPBs belonging to two cores. We compare the performance of our library against Intel's standard RCCE library. The data transfer latency between cores depends upon the data size as well as the distance in terms of the number of hops between them because the library uses the on-tile MPB for communication.



(a) Core 0 to Core 2 (1 hop)

(b) Core 0 to Core 4 (2 hops)

(c) Core 0 to Core 6 (3 hops)

(d) Core 0 to Core 8 (4 hops)

(e) Core 0 to Core 10 (5 hops)

Fig. 8: Inter-core communication latency for payload sizes between 32 and 4096 bytes using Intel's RCCE library and our library (LWRR, RWLR). The latency increases linearly (note that the x-axis is plotted using logarithmic scale) with payload size and the number of hops.

Therefore, in our experiments, we studied data transfer latency for cores that were 1 to 5 hops apart as shown by the greyed out boxes in Figure 6.

Figure 8 shows the results of our experiments for data sizes varying between 32 bytes and 4096 bytes. Each data point in these plots is an average over 90 measurements. We note that the variance in these values is very small and not visible in the plots. We see that the data transfer latency increases linearly with payload size for both our library and the standard RCCE library. The transfer time also increases with the increase in distance between cores. Furthermore, our library performs better than the RCCE library due to its asynchronous nature, i.e., in our library the sender does block until the receiver acknowledges the transfer on every data exchange before continuing with other tasks. However, the sender waits until the previous message is flagged as read before transmitting a new message.

In summary, our library is not only more secure for communication between the security kernel and slave applications but also exhibits better performance than the standard RCCE library.

VII. Related Work

To the best of our knowledge, there is no work that directly addresses the problem of securing the execution of software on many-core architectures. We believe that this is partly due to the fact that such architectures have primarily been proposed and designed for high-performance computing via fast memory interconnects between different cores. Previous attempts that come the closest to our work are from the domain of safety engineering which try to isolate faults [21] and provide timing guarantees through spatial and temporal isolation of accesses to shared resources [4]. However, these papers do not consider any compromise of system components (including their equivalent of the TCB) and hence do not address the security related issues as we do here.

Many-core systems: Most research on many-core systems so far, has focused on improving the performance of high performance computing [22], [23] or on implementing traditional OS functionality [20], [24], [25]. More specifically, in [20], the authors present a framework for bare-metal execution on Intel's SCC. Similarly, in [24], the authors propose a framework to execute a bare-metal hypervisor on such platform and this is extended in [25] to enable fast inter-kernel communication running on the bare-metal hypervisor. In contrast to these proposals, our work does not focus on improving the performance on many-core systems but instead tries to improve their security. Furthermore, unlike other existing work, our prototype enables the co-existence of bare-metal and full-fledged operating systems running securely on many-core platforms.

Porting kernels to Intel's SCC: In [15] the authors propose a way to port Barrelfish, a multikernel architecture [26] to Intel's SCC. Similarly, in [16] the authors provide a port of the microkernel Fiasco.OC to Intel's SCC. In both cases, the authors point out that memory isolation is not currently possible on the SCC and mention that they intend to address it as part of future work.

However, currently, as far as we can tell, these papers do not address the various security aspects of many-core systems as we do here.

Security Using Networks-on-Chip(NoC): There have been a number of proposals in the past to leverage the NoCs on chips to enforce security. For example, there have been proposals to enhance NoCs to perform access control [4], [27], [28], monitor system behavior using NoCs [29] as well as to protect communication between cores [30]. However, none of these papers discusses any mechanisms to protect against a compromised security kernel or context awareness as we define it in this paper.

Security kernels and hypervisors: Research in securing operating systems has been traditionally carried out by shrinking the trusted computing base of a system. This has led to the development of microkernels that have been proposed and deployed in the context of personal computers [31] and embedded systems [5]. Traditionally, a microkernel in contrast to a monolithic kernel only supports memory isolation, inter-process communication and threading. All other services (e.g., file systems, memory management) run as user-space tasks. On currently available many-core systems, such microkernel solutions would have to execute on every core to control access to resources.

In cloud computing environments where multiple clients want to share the same platform (usually through virtualization), a thin layer of software called the hypervisor manages access to system resources [32], [33]. More recent work in particular on NOVA [6] has led to new solutions that further reduce the code-base of a hypervisor which constitutes the TCB in virtualized systems. The main ideas of a microhypervisor, such as NOVA, have been borrowed from traditional microkernels. As in the case of microkernels, such a hypervisor would have to run on every core in a many-core system to guarantee isolation of software running on it.

In addition to reducing the size of the system TCB itself, there have also been efforts to minimize its interaction with other software components through disengagement. However, in contrast to existing disengaged solutions like NoHype [14], which require static resource allocation, our architecture allows applications to dynamically request for more resources without a large increase in complexity as we showed in our prototype.

To summarize, in contrast to related work, we have analyzed the support for security that exists in commercial many-core systems, defined new security properties like context awareness that they can potentially support and explored the design space for secure many-core systems in general and in the context of Intel SCC in particular.

VIII. Conclusion

In this work we explored the feasibility of creating and managing isolated execution environments on many-core systems. We showed that, on commercial many-core platforms, one has to run a trusted agent (TCB element) on every core to enforce isolation between applications in different execution environments. We then presented SEMA, an Intel SCC-based architecture that leverages the Look-Up-Tables (LUTs) available for each core to control access to all system resources. In SEMA, only the security kernel can modify the LUTs of all cores and, therefore, applications cannot circumvent the protections configured by the kernel through the LUTs.

Although SEMA uses a small security kernel and hence faces a low risk of its compromise, we still propose new mechanisms to defend against a malicious kernel. We defined a new security property called context awareness which refers to the ability of individual applications being able to learn and discover about what else in the system is configured to interact with them without the help of the kernel. Furthermore, we implemented SEMA on the Intel SCC and evaluated the performance of loading and executing a Linux OS from a small kernel. We also implemented a new communication library for data exchange between the kernel and the other cores in the system. Our work shows that existing many-core architectures with small modifications can provide strong security guarantees and also limit the capabilities of the system's security kernel; this makes them suitable for use in several applications ranging from cloud and mobile computing to safety-critical systems.

References

[1] Adapteva, "Ephiphany Multicore IP," www.adapteva.com/products/epiphany-ip/epiphany-architecture-ip/, last access 2013.

[2] Intel Corporation, "SCC External Architecture Specication (EAS)," https://communities.intel.com/servlet/JiveServlet/previewBody/5852-102-1-9012/SCC_EAS.pdf, Revision 1.1, last access 2013.

[3] R. Jayaram Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun, "Enabling Trusted Scheduling in Embedded Systems," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC'12, 2012, pp. 61–70.

[4] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality," in *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, 2012, pp. 24–31.

[5] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP'09, 2009, pp. 207–220.

[6] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, 2010, pp. 209–222.

[7] Y. Zhang, W. Pan, Q. Wang, K. Bai, and M. Yu, "HypeBIOS: Enforcing VM Isolation with Minimized and Decomposed Cloud TCB," www.people.vcu.edu/~myu/s-lab/publications/Zhang2012.pdf, Tech. Rep., 2013.

[8] Intel Corporation, "Intel Trusted Execution Technology Measured Launched Environment Programming Guide," www.intel.eu/content/www/eu/en/software-developers/intel-txt-software-development-guide.html, last access 2013.

[9] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP'13, 2013.

[10] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP'13, 2013.

[11] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP'13, 2013.

[12] Advanced Micro Devices, "AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual," http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf, last access 2013.

[13] ARM, "Building a Secure System using TrustZone Technology," http://www.arm.com, 2009.

[14] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "NoHype: Virtualized Cloud Infrastructure Without the Virtualization," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA'10, 2010, pp. 350–361.

[15] S. Peter, T. Roscoe, and A. Baumann, "Barrelsh on the Intel Single-chip Cloud Computer," www.barrelfish.org/TN-005-SCC.pdf, 2013.

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

[17] Tilera Corporation, "TILEPro Processor Family," www.tilera.com/products/processors/TILEPro_Family, last access 2013.

[18] ——, "Tile Processor Architecture Overview for the TILEPro Series," www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf, release 1.2, last access 2013.

[19] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. van Doorn, "CARMA: A Hardware Tamper-resistant Isolated Execution Environment on Commodity x86 Platforms," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12, 2012, pp. 48–49.

[20] M. Ziwisky and D. Brylow, "BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, Toulouse, France, 2012.

[21] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: Building high availability systems with commodity multi-core processors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, 2007, pp. 470–481.

[22] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance Analysis and Benchmarking of the Intel SCC," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 139–149.

[23] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, 2011, pp. 525–532.

[24] P. Reble, J. Galowicz, S. Lankes, and T. Bemmerl, "Efficient Implementation of the bare-metal hypervisor MetalSVM for the SCC," in *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, Toulouse, France, 2012.

[25] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl, "A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM," in *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[26] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP'09, 2009, pp. 29–44.

[27] S. Lukovic and N. Christianos, "Enhancing Network-on-chip Components to Support Security of Processing Elements," in *Proceedings of the 5th Workshop on Embedded Systems Security*, ser. WESS '10, 2010, pp. 12:1–12:9.

[28] J.-P. Diguet, S. Evain, R. Vaslin, G. Gogniat, and E. Juin, "NOC-centric Security of Reconfigurable SoC," in *Proceedings of the First International Symposium on Networks-on-Chip*, ser. NOCS '07, 2007, pp. 223–232.

[29] L. Fiorin, G. Palermo, and C. Silvano, "MPSoCs Run-time Monitoring Through Networks-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09, 2009, pp. 558–561.

[30] C. Gebotys and R. Gebotys, "A Framework for Security on NoC Technologies," in *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, 2003, pp. 113–117.

[31] J. Liedtke, "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, 1995, pp. 237–250.

[32] Xen Project, "The XEN Project," http://www.xenproject.org/, last access 2013.

[33] VMware, "VMware Incorporated," http://www.vmware.com/, last access 2013.