Information Technology

Sicherheit bei Bedarf durch Invasives Rechnen

Providing Security on Demand Using Invasive Computing

Gabor Drescher: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 4, Martensstr., 1, D-91058 Erlangen, Germany
E-Mail: drescher@cs.fau.de
Gabor Drescher is a doctoral researcher at the chair for distributed systems and operating systems at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Christoph Erhardt: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 4, Martensstr., 1, D-91058 Erlangen, Germany
E-Mail: erhardt@cs.fau.de
Christoph Erhardt is a doctoral researcher at the chair for distributed systems and operating systems at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Prof. Dr.-Ing. Felix Freiling: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 1, Martensstr., 3, D-91058 Erlangen, Germany
E-Mail: felix.freiling@cs.fau.de
Felix Freiling is a professor of computer science at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Johannes Götzfried: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 1, Martensstr., 3, D-91058 Erlangen, Germany
E-Mail: johannes.goetzfried@cs.fau.de
Johannes Götzfried is a doctoral researcher at the chair for IT-Security Infrastructures at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Dr.-Ing. habil. Daniel Lohmann: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 4, Martensstr., 1, D-91058 Erlangen, Germany
E-Mail: lohmann@cs.fau.de
Daniel Lohmann is an associate professor (PD) at the chair for distributed systems and operating systems at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Pieter Maene: KU Leuven, Department of Electrical Engineering (ESAT), COSIC, Kasteelpark Arenberg, 10, B-3001 Heverlee, Belgium
E-Mail: pieter.maene@esat.kuleuven.be
Pieter Maene is a doctoral researcher at the COSIC research group at KU Leuven, Belgium.

Dr.-Ing. Tilo Müller: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 1, Martensstr., 3, D-91058 Erlangen, Germany
E-Mail: tilo.mueller@cs.fau.de
Tilo Müller is a post-doctoral researcher at the chair for IT-Security Infrastructures at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Prof. Dr. Ir. Ingrid Verbauwhede: KU Leuven, Department of Electrical Engineering (ESAT), COSIC, Kasteelpark Arenberg, 10, B-3001 Heverlee, Belgium
E-Mail: ingrid.verbauwhede@esat.kuleuven.be
Ingrid Verbauwhede is a professor of electrical engineering at the COSIC research group at KU Leuven, Belgium.

Andreas Weichslgartner: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 4, Cauerstr., 11, D-91058 Erlangen, Germany
E-Mail: andreas.weichslgartner@fau.de
Andreas Weichslgartner is a doctoral researcher at the chair for Hardware/Software Co-Design at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Dr.-Ing. Stefan Wildermann: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Lehrstuhl für Informatik 12, Cauerstr., 1, D-91058 Erlangen, Germany
E-Mail: stefan.wildermann@fau.de
Stefan Wildermann us a post-doctoral researcher at the chair for Hardware/Software Co-Design at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.

Keywords: Security and privacy - Embedded systems security; Security and privacy - Operating systems security; Security and privacy - Information flow control; Social and professional topics - Computer crime

Abstract

The *invasive computing* paradigm offers applications the possibility to dynamically spread their computation in a multicore/multiprocessor system in a resource-aware way. If applications are assumed to act maliciously, many security problems arise. In this acticle, we discuss different ways to deal with security problems in a resource-aware way. We first formalize the attacker model and the different security requirements that applications may have in multi-core systems. We then survey different hardware and software security mechanisms that can be dynamically configured to guarantee security on demand for invasive applications.

Zusammenfassung

Invasives Rechnen bezeichnet ein neues Programmierparadigma, das es parallelen Programmen erlaubt, sich ressourcengewahr auf einem Multicore/Multiprozessor-System auszubreiten und die verwendeten Ressourcen nach Abarbeitung der Aufgabe wieder freizugeben. Bei bösartigem Programmverhalten, etwa bei konkurrierenden Ressourcenanforderungen, entstehen Sicherheitsprobleme, die durch entsprechende Mechanismen zur Laufzeit behandelt werden müssen. Dieser Beitrag gibt einen Überblick über derartige Sicherheitsmechanismen, die Sicherheit bei Bedarf durch Invasives Rechnen ermöglichen.

1 Introduction

Invasive computing [31] is a systems paradigm that focuses on leveraging the computing power provided by future multi- and manycore systems. Its name stems from the notion that applications are written in a resource-aware manner and can dynamically acquire (*invade*) resources according to their needs for computation power, communication bandwidth/latency or memory size. The principal idea is that applications are granted exclusive access to these resources, which enables them to optimally adapt themselves to the resources they are holding. In such a system, applications compete with each other for a share of the hardware resources.

If applications can act maliciously, various security-related issues arise. The key concepts of invasive computing, namely resource invasion, dynamic resource management, heterogeneous multicore technology, and network-on-chip interact with security considerations in multiple ways: On the one hand, exclusive access to resources *prevents* many security problems if exclusiveness is implemented correctly (preventing unauthorized accesses towards providing the security properties of integrity and confidentiality). On the other hand, however, exclusiveness is in conflict with the security property of availability, and resource awareness is known to create side channels that endanger confidentiality. In this paper we show how invasive computing and security can be reconciled. We argue that this can only happen if security is taken into account *during design* of invasive systems and not as a patch later on, and that security is treated equally at all architectural design layers. Indeed, security and trust can only be provided if there is a chain of trust from the user application down to the hardware component. We therefore survey concepts which the authors have developed to address security issues in invasive computing over recent years and which cover the three main architectural levels (see Table 1).

layer	**issues**
application	specifying security properties at the programming-language level; translation of security requirements to invasive resource constraints
operating system	run-time enforcement of basic confidentiality and integrity properties; information-flow protection through resource partitioning
hardware	establishing minimal Trusted Computing Bases (TCBs) in hardware dynamically; isolating applications within the Network-on-Chip (NoC)

Table 1: Overall layered organisation and issues at each layer.

At the application layer, we first need to define what we mean by "security" in this novel context. This results in security requirements which we discuss in Section 3. Security partitioning, also called *isolation*, is our main method to obtain security. Isolation as a *security mechanism* exists at very different abstraction layers. We first consider basic issues of software isolation at different levels of abstraction in Section 4. There is also hardware isolation at different levels of granularity, e.g., different physical cores connected by a network-on-chip or different components in one tile. We discuss this in Section 5.

For readers new to the topic, we give a brief introduction to the necessary background next.

2 Background

The principal driver behind the idea of invasive computing is the insight that the current trend towards integrating several dozens up to hundreds of cores onto a single chip is not going to stop anytime soon. Invasive computing is an effort with the goal of mastering the challenge of how to remain scalable in a world of extreme potential for parallelism. Invasive computing is a comprehensive approach that covers all layers from the application software down to the hardware.

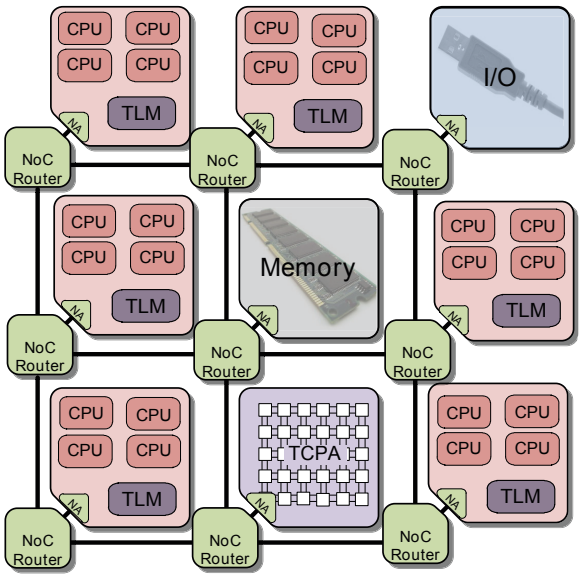


Figure 1: Example of a tiled invasive hardware architecture.

2.1 Invasive Hardware

On the hardware level, the current approaches to invasive computing use a tiled architecture such as the one shown in Figure 2. A tile comprises a number of CPU cores that are interconnected through a common bus and share a slice of fast on-chip tile-local memory (TLM). Tiles are interconnected with a two-dimensional NoC. There are a number of special tiles that house off-chip DDR memory, IO ports, or accelerator units such as a tightly coupled processor array (TCPA) [14]. While all cores within a tile share a coherent view on memory, no cache coherency is provided across tile boundaries; instead, message-based communication over the NoC has to be performed. For the purpose of predictability and security, tiles can be allocated exclusively to single applications as a basic constructive means for temporal and spatial isolation.

2.2 Invasive (Systems) Software

OctoPOS [20, 21], the invasive parallel operating system, is the entity that makes the hardware available to applications, enforcing their resource-allocation requests. It works similarly to a (closely coupled) distributed system in the sense that a separate OS instance exists per tile. Invasive applications that span several tiles have to reflect this fact. Applications can be written in a traditional language like C or C++, but for a more natural view on the underlying architecture, a specialized language with a notion of a partitioned global address space (PGAS) model is preferable. Primary support exists for InvadeX10 [7, 8, 19], an extension of the type-safe X10 language for distributed programming that draws inspiration from Java and Scala.

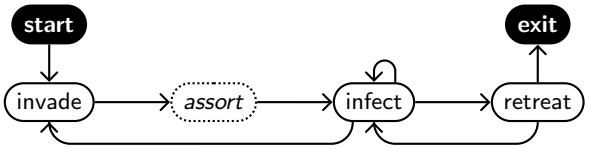


Figure 2: Fundamental structure of an invasive program.

Figure 2 shows the application flow of an invasive program. First, during the *invade* phase, the application expresses a request for a set of resources (such as processing elements) to the run-time system. An invasion request can express additional non-functional requirements such as a need for confidentiality. The system then decides, based on the global system state and all additional constraints it was given, which resources to assign to the application, and returns a *claim* describing these resources. Multiple applications can compete for the same set of resources, in which case the returned claim may contain only a subset of what the application requested. In the optional *assort* phase that follows, the application can adapt itself according to the contents of the claim, for example by choosing an appropriate algorithm. The claim can then be *infected*, which leads to the parallel execution of application code on the claim's processing elements. After the execution has finished, the application may either reuse and adapt the claim for further computations or *retreat* from it, releasing the associated resources.

The principle of invasion is not limited to CPU cores, but can be applied in an analogous manner to other resource classes such as memory regions, NoC channels, or accelerator units. In all cases, an application-driven exclusive reservation of resources is performed, enabling an application to tune itself according to the concrete set of resources that were made available to it.

2.3 Attacker Model

No definition of security makes sense without stating the attacker model which is assumed. Since the main application areas currently envisaged for invasive computing are computing centers [10, 24], it is argued that the invasive hardware can be sufficiently well protected. The attacker behavior can therefore currently be restricted to "software-only" attacks, meaning that the attacker is able to inject arbitrary code at different levels of the system: (A1) at the level of X10 programs, (A2) as binary code at application level, (A3) at the operating system level. So while *physical* attacks are *not* considered, we assume that the attacker may also in the worst case compromise the systems software. Overall, since remote exploitation of commodity systems is far more common than exploitation through physical access, we believe that this is a reasonable assumption [29].

Given the software-only attacker, we consider attacker goals that inhibit the classical security properties

of confidentiality and integrity – that is, we consider stealing of confidential code or data through direct access or indirect means (such as side channels) and the targeted modification (infection) of other applications. For brevity, achieving the security property of availability, i.e., dealing with denial of service attacks through resource monopolization, is excluded from explicit focus in this paper.

3 Specifying Security Requirements

To be called secure, the invasive computing system must provide security mechanisms to ensure certain security requirements given the attacker model sketched above. In this paper, we focus on the classical security properties of confidentiality and integrity that must be ensured in the presence of untrustworthy programs that compete for resources and can contain malicious functionality. Broadly speaking, confidentiality refers to the absence of unauthorized information disclosure while integrity refers to unauthorized system manipulation. The reference for these properties are different information domains, usually defined by the information contained within and managed by a software application.

We now present confidentiality and integrity requirements that have been formalized in the context of invasive computing and can be requested by invasive applications within the invade phase. Intuitively, a programmer should require a form of confidentiality if the contents of the program (code, data, computation) are to be kept confidential, whereas a programmer should require a form of integrity if the computation is to proceed without sabotage. If the invasive runtime system returns a suitable claim, the security requirements are guaranteed between the infect and retreat phases through all levels of the architecture. Since tolerating an attacker at the X10 level is obviously less resource-intensive as at the operating-system level, a crucial ingredient for the defense mechanism is at which level the attacker is expected. Overall, this can only be determined by the computing center providing the invasive computing architecture and only seldomly can be specified by the application programmer. The security mechanisms described later are able to handle attackers down to the operating system level.

3.1 Confidentiality

We provide two forms of confidentiality: (basic) confidentiality (C) and ϵ-confidentiality (ϵC). Requesting ϵC for any ϵ implies C.

The difference between C and ϵC is delicate. For most practical situations, requesting C suffices as it corresponds to the "classical" understanding of confidentiality as provided by standard runtime protection techniques in operating systems (memory protection). The requirement of ϵC intuitively protects as much as C, but furthermore provides "state of the art" protection against side-channel attacks. If requesting ϵC, the programmer has to specify a value ϵ which is the maximum acceptable rate of information flow in bits per second (bps). The formal definition of ϵC is as follows:

Definition 1 (ϵ-confidentiality, ϵC)**.** *The invasive software S satisfies ϵC for attacker A and environment E iff there exists evidence of attacks on S by A in E that lead to unauthorized information leakage of at most ϵ bits per second (bps).*

In the definition, S refers to a concrete hardware/-software configuration of an invasive application, A refers to the different types of software-only attackers from the attacker model, and E specifies all other environmental circumstances outside of S. Intuitively, the definition can be rephrased as follows: "S satisfies ϵC iff the best published attack on S achieves ϵ bps." This is how the definition takes the "state of the art" into account, it is therefore relative to the knowledge of existing attacks on S. Therefore, a system that satisfies ϵC for $\epsilon = 5$ bps today might not satisfy this property anymore if a new attack on S becomes known that achieves an information leakage of 50 bps. If the latter is the best possible attack today, then the system satisfies ϵC with $\epsilon = 50$ bps. Theoretically, if nobody has attacked the system yet, it satisfies ϵC for $\epsilon = 0$ bps. This shows that ϵC is only meaningful if many people have tried very hard to attack S and have published their results. Our definition therefore corresponds to security definitions of cryptographic primitives that also depend on how well researched a primitive is. Note that the value ϵ can also be understood as an inverse metric to the "effort" that is currently considered to break the system. An application programmer who is considering requiring ϵC should therefore not take ϵ too literally but more as a qualitiative measure. If the best possible attacks are 50 bps, then the programmer should request ϵC for $\epsilon = 100$ bps, for example, to achieve at least some margin. If the field of invasive security has stabilized (and invasive architectures have been attacked for 50 years without reaching more than 50 bps), the confidentiality guarantee of $\epsilon = 100$ bps is substantially meaningful.

The notion of ϵC refers to any unauthorized data flow from S to the outside, it therefore aims at applications with very high security guarantees (military) and is inspired by Lampson's observations on the confinement problem [16] and by formal information-flow concepts such as non-interference and its many flavors [12, 17]. While C — like ϵC — refers to the absence of unauthorized information flow, it refers to notions of information flow that can be detected at runtime by observing an individual system trace. It is well-known that side channels are in general not detectable by observing a single system trace [26]. Our definition of C therefore refers to

all types of confidentiality that can be expressed as sets of traces [4], more specifically as "safety properties", i.e., trace properties that are violated in finite time.

Since ϵC implies C, $\neg C$ also implies $\neg\epsilon C$. To ensure ϵC it is therefore necessary to provide C first, which has also been the main research focus until now. This explains why the contributions that we describe in this paper aim to satisfy C.

3.2 Integrity

There are two forms of integrity: (basic) integrity (I) and integrity with attestation (Iwa). Reqesting Iwa implies requesting I.

The property of I refers to the classical notion of integrity in the sense that a program can be sure that code and data have not been modified in an unauthorized manner. The difference between I and Iwa refers to the possibility of proving to a third party that integrity has been satisfied. This is generally known as "attestation" in the literature [9]. Iwa is interesting for applications that are deployed on remote computing centers in which the physical environment or the system maintainers might be untrustworthy (as is the case in many "cloud" contexts). Using Iwa, the application receives a special token (the attestation or proof) which it can send to a third party that can verify integrity.

3.3 Satisfying Security Requirements at Runtime

At the application layer, the overall goal is to offer applications a certain amount of protection. However, applications need to specify their security requirements using elements of the programming language X10. A security requirements is specified by a level of confidentiality and integrity requested by the application and are embedded into the constraint formalism used to specify the invasive claim. This includes "global" requirements (such as ϵC) or requirements at the interface, e.g., which inputs to the program (ports) are confidential and which ones are not. This even allows for information-flow control *within* an application, which is investigated elsewhere [28, 25].

At compile-time, the requirements are processed into a format that the application run-time system can deal with and dynamically adjust the security mechanisms at runtime on demand so that the requirements are satisfied depending on the attacker model A1–A3. This can involve different placement strategies as well as hardware and security mechanisms. All of these mechanisms are discussed in the following sections.

4 On-demand Security Mechanisms in (Systems) Software

We now turn to the operating system, which is responsible for introducing and maintaining basic confidentiality and integrity requirements, implemented using isolation mechanisms. Generally speaking, isolation comes in two dimensions: spatial and temporal.

On the one hand, *spatial isolation* prevents applications from interfering with each others' state and is an essential precondition for both confidentiality and integrity. It is usually ensured through memory protection, which is enforced by hardware with OS support. The technique commonly implemented by general-purpose operating systems, such as Linux, is isolation through address-space virtualisation. Other systems, in particular in the embedded field, may choose a more light-weight MPU-based approach.

On the other hand, *temporal isolation* ensures that applications cannot interfere with the temporal constraints of other applications, for instance by monopolizing the CPU. This can be achieved through time-sharing, where the OS multiplexes the CPU among applications. However, time-sharing is potentially problematic for full confidentiality: It may allow opening a side channel from another application by observing that application's temporal or caching behavior.

To guarantee that the operating system is the sole entity that can manage spatial and temporal isolation, an additional type of isolation is needed: *privilege isolation*. Privilege isolation protects OS functionality from being taken over by unprivileged, possibly untrustworthy code, which would undermine all security guarantees.

4.1 Constructive Measures

By design, the application model of invasive computing makes it easier for the system to provide isolation than for ordinary systems. Since CPU cores are generally not shared among applications, there is no need for explicit temporal isolation and side channels are naturally restricted. However, information can still flow over physical channels such as heat dissipation, system load, cache-hit rates or network congestion, which can be queried by resource-aware programs and which endanger ϵC. Such side channels can be further avoided through stronger spatial isolation, in which, for example, a set of unused processing elements or memory banks is left free between two applications on a tile. These buffer zones correspond to "physical borderlines" (or "dark resources") that further decrease the potential of side channels such as heat dissipation [22]. A sensitive application may even ask the run-time system to grant exclusive access to an entire tile.

The system is responsible for placing applications in such a way that their security constraints are fulfilled.

As detailed in Section 5.2, our mapping strategies to achieve this use a hybrid approach that employs both offline analysis at design time and online mapping at runtime.

Spatial isolation is implemented in OctoPOS through memory protection, and privilege isolation is implemented through hardware protection rings – but as we will point out in the following section, these kinds of protection are actually not always necessary.

4.2 The Cost of Memory Protection

Memory protection is fundamental for software safety and security when dealing with arbitrary applications from potentially untrustworthy origins. It stands to reason that contemporary general-purpose operating systems enable this protection by default. However, memory protection also adds substantial time overhead to operating-system services:

Since memory protection requires privilege isolation, system calls must be implemented as traps with mode switches. This entails saving (and later restoring) parts of the processor state, and adversely affects caches and the pipeline.

However, the even costlier part of memory protection comes whenever a massively parallel application dynamically changes its memory mappings. Beside the obvious operations on the page tables themselves, Translation Lookaside Buffer (TLB) entries must be invalidated or even the TLB must be emptied completely. This causes TLB misses in the further execution of application code, which lead to increased memory-access times. When a parallel application unmaps a memory page from its address space, it must not only remove the respective entry from the page table and update its own TLB, but also the TLBs of all other CPUs belonging to that application. This is the so-called TLB-shootdown operation [6]. It is a rather expensive operation since all the CPUs have to be visited that belong to the application to invalidate certain TLB entries. After the TLB shootdown, it is guaranteed that no application thread can access the unmapped memory region.

State-of-the-art systems only support memory protection for all applications, for none of them (embedded systems), or statically partitioned for some software components. However, these general protection mechanisms may be superfluous in certain cases. Programs that are written in a type-safe language using a trustworthy run-time system can only access their own objects and are unable to modify arbitrary memory locations. Here, memory protection does not provide significant benefits for security. Programs for the invasive architecture written in X10 represent this kind of type-safe software. Furthermore, we have trustworthy applications with a high demand for predictability or performance for which we also want to disable memory protection and the associated costs.

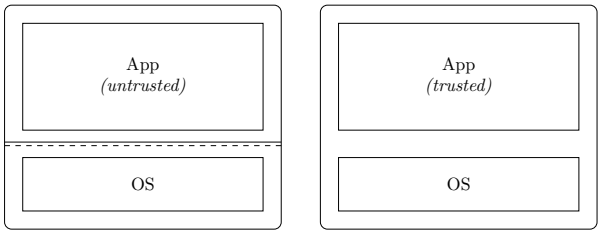
The decision when to enable or disable memory protection, and for whom, is made by the operating system at run time whenever isolation demands change (for instance, if a new application arrives). This decision, however, potentially affects all applications on the tile. This is the case because by means of memory protection, an application cannot protect itself from malicious accesses by others – it can only ask the OS to restrict *all others* from reaching outside of their own memory areas. Consequently, whenever one application needs protection, everybody else within that tile gets to pay.

The traditional *defense-in-depth* principle states that multiple layers of security are to be preferred over a single point of failure: hardware-based isolation should always be available as a fallback solution in case the software-based isolation fails, for example due to bugs in the compiler or the runtime system. We argue that there is a significant trade-off between the costs and benefits of in-depth defence [2], which is why we leave this choice to the user. Applications with high security demands can request to receive additional hardware-based isolation from other applications, even if the latter are type-safe per se. On the other hand, applications with a less strict need for security can choose to abstain from enforcing hardware protection.

Figure 3 shows a number of possible isolation scenarios within a tile. In the simplest case, there is only a single application, which is isolated from others by definition. Under regular circumstances, i.e. attacker model A2, as in Figure 3(a), privilege isolation is ensured through technical means (i.e., system calls). However, given attacker model A1, the application is certifiably trusted (e.g., written in a type-safe language such as X10) and, as shown in Figure 3(b), the operating system does not even need protection from the application to ensure that the system state cannot be corrupted.

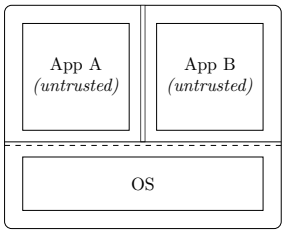
In case of more than one application and given attacker model A2, as can be seen in Figure 3(c), the untrusted applications are isolated from both the operating system (vertically) and all other applications (horizontally). Trusted applications – see Figure 3(d) – can still continue running unrestricted, without security measures enforced by the operating system.

Finally, Figure 3(e) shows an untrusted and and a trusted application running together on the same tile. In this case, only the untrusted application has to pay the price for memory protection.

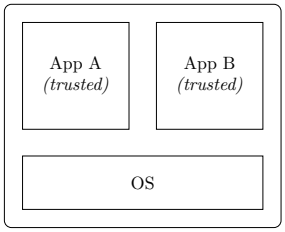


(a) Single untrusted app

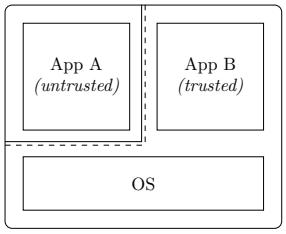
(b) Single trusted app



(c) Multiple untrusted apps



(d) Multiple trusted apps



(e) Untrusted and trusted app side-by-side

Figure 3: Different application scenarios of spatial isolation within a tile. A solid double line denotes isolation in both directions; a broken line on one side indicates that no isolation is enforced for accesses from that side.

4.3 Implementing Adaptive, Dynamic Memory Protection

We implement a dynamic protection scheme in OctoPOS that allows the OS to enable and disable protection for multi-core applications on the invasive architecture. This enables the OS to adapt to the current runtime situation and to provide just the right amount of protection where necessary, and to leave trustworthy applications unisolated.

To facilitate such an adaptive protection, we track the memory regions belonging to an application. These regions include the static text and data sections as well as regions assigned by dynamic memory allocation. It is important to note that dynamic memory regions are only assigned to an application when invading resources and withdrawn when reinvading or retreating from resources. This happens in a much lower frequency than the usual `malloc/free` calls that just partition the already assigned dynamic memory regions in user space. With this region information at hand, turning on memory protection entails building a page-table hierarchy and applying these page tables on all CPUs currently belonging to this application. The inverse operation removes the page-table mapping from each application CPU. Depending on the hardware architecture, the MMU can be disabled or, as it is the case on x86_64, a simple mapping is activated that grants access to all user-space memory regions. Consequently, trustworthy multi-threaded applications that already have access to all local memory do not cause page-table updates, TLB invalidations or TLB shootdowns.

# cores	protection		
	on	**off**	**not available**
1	3425	388	82
2	3794	387	83
3	5376	387	83
4	6480	388	82

Table 2: Clock cycles for the reinvade operation, applications spanning one to four CPU cores.

We measured the execution time for important system services including invade, retreat and claim construction for protected and unprotected applications, as well as a configuration of our OS that does not support memory protection at all and does not track application-memory ranges (and here serves as a baseline for the costs of memory protection). Our test platform was an Intel Xeon E3-1275 quad-core system running at 3.5 GHz. Cycles were measured using `rdtsc(p)`. Table 2 shows the measured CPU cycles for memory-resource expansion and contraction (reinvade). Page tables have to be updated in the protected case (column *on*) and this change is then propagated via inter-processor interrupts to all other cores that belong to the application. This event propagation is a synchronous operation, since the changes have to be applied before the application has a chance to erroneously use memory regions. Therefore we see a linear increase in cycles for the protected case. When protection is available, but currently turned off (column *off*), only some bookkeeping takes place to register or de-register memory regions for the application. The variant without any memory protection even omits this bookkeeping as the application may access all memory anyway. Compared to the *on* mode (which is the only available mode in standard operating systems), the runtime of this operating-system service can be reduced by factors of 8.8 to 16.7 by omitting memory protection when not needed (*off* mode). However, it is not just the run time that can be lowered for unprotected applications, but we also improve predictability, since the runtime of an invade, retreat or reinvade does not strongly depend on the number of cores that the application currently utilizes. Furthermore, application cores that are currently in computation are not interrupted to do expensive TLB invalidations when some other application core invokes an OS service.

When memory protection is switched *on*, system calls are implemented as traps. Otherwise, the application runs in privileged mode, so a system call is effectively a simple and very efficient library call without additional overhead.

4.4 Cross-Tile Concerns

All the operations presented above take place within a single cache-coherent tile. However, applications can span multiple tiles and a communication mechanism needs to be established that also complies with memory-protection rules.

Our hardware platform provides a high-speed and asynchronous DMA unit for low-level communication between tiles using the NoC. We intended to leverage the performance of this unit to enable asynchronous user-space communication in a secure and flexible way, even in the face of variable protection states of the same application on different tiles. This unit does not obey any memory-protection rules as defined in the page tables. For this reason, we need to enforce memory protection across tiles in software by the operating system when using the DMA engine, as shown in Figure 4.

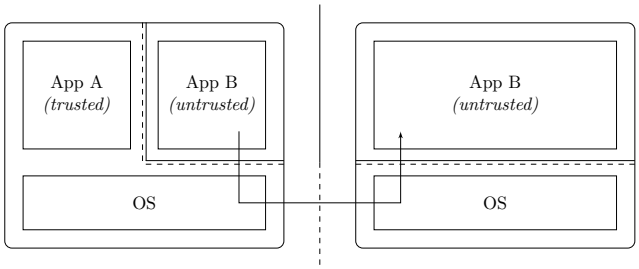


Figure 4: Cross-tile spatial isolation.

A simple implementation would visit the target tile, check access rights, and return positive or negative feedback to the sender, discarding the operation if access rights do not allow writing to the specified target region. This operation would take place on every message sent and is quite costly as we will show later. Instead, the implementation consists of an operating-system-level distributed software-cache mechanism to cache and invalidate access rights for remote memory regions. New entries are placed into the local cache after visiting the target tile and examining the currently active protection state and access rights. This information is then returned to the sending tile, which creates a new cache entry or expands an existing entry. The target tile in turn saves the information that a cached entry exists on the sending tile. Further DMA transfers to the same memory region just do a lookup in the local cache without the overhead of doing expensive checks on the remote tile. As memory regions change by retreating or reinvading memory resources, cached entries may lose validity. In these cases, invalidations are sent to tiles that may be holding descriptors for the respective memory region in their caches. We assume that these invalidations happen at a much lower frequency than message-passing operations.

With this cache in place, we were able to provide a DMA-like message-passing mechanism to applications. This mechanism is also asynchronous without violating the protection rules defined by the memory mapping. To evaluate the performance impact of the software-managed message-passing mechanism, we ran the NAS Parallel Benchmarks suite [5]. These standard benchmarks provide programs written against the MPI programming interface that exhibit different communication patterns. To run the unmodified benchmarks, we implemented a custom MPI library on top of our low-level message-passing mechanism. Run times with memory protection and enabled cache are 0.2–2.0 percent slower than in the unchecked and unprotected variants. Naively querying the access rights for every message on the remote side adds between 20 and 200 percent to the total run time of the benchmarks.

4.5 Summary

In summary, our operating system OctoPOS makes use of the architectural characteristics of the hardware platform to efficiently provide spatial, temporal and privilege isolation. The tiled invasive architecture is a good fit for this. However, hardware-based memory protection does induce significant overhead and is not needed to isolate type-safe programs. Thus, we enable those protection measures only when actually necessary, following the fundamental design principle that "'less demanding' applications should not be forced to pay for the resources consumed by unneeded features" [23] also with respect to isolation.

5 On-demand Security Mechanisms in Hardware

To guarantee security properties, software-based security mechanisms like those presented in the previous section can be used if the attacker merely is able to inject code at application level. However, if system software is considered untrusted as well (attacker model A3), implementing security mechanisms in software is not sufficient anymore and hardware support is needed instead to protect against, for example, a malicious operating system.

This area of secure execution environments was pioneered by Suh et al. [30] and has evolved into readily available hardware extensions such as SGX [18]. In this section, we survey our results that aim at lightweight embedded devices and possess a hardware-only TCB at

the same time. We first describe the possibility of guaranteeing confidentiality, integrity and isolation through hardware-supported architectures with minimal TCBs and, secondly, present security mechanisms to ensure those principles for NoCs.

5.1 Hardware-Supported minimal TCBs

First, we discuss a solution which addresses code and data confidentiality, effectively protecting developers against intellectual-property loss, amongst others. However, any security architecture aimed at protecting invasive computing systems has to be scalable. A scalable data-isolation mechanism built on strong hardware-based encryption is therefore presented afterwards.

5.1.1 Program-Counter-based Memory Access Control

Our first approach [13] protects an application's code and data by means of program-counter-based memory-access control, i.e., for each application certain memory-access restrictions exist which are enforced by the CPU. Our solution has a small hardware-only TCB, and uses a minimal number of hardware features. The system, however, makes no guarantees on availability.

The protected memory region of each application is divided into two sections, one for code and constants and one for protected data. The boundaries of these regions are stored in dedicated registers which are added to the processor architecture. These registers are used as inputs to the memory-access logic which compares them to the current program counter to enforce the access rights. The code in the text section can *only be executed* when the program counter is either at the entry point or the application is already executing. The application's code and data can only be read or written when the program counter is in the application's text section.

To secure an application's code before being loaded, we decided to store it encrypted within memory and to design a separate loader application. This loader application is responsible for decrypting and protecting the code. The decryption key for the application is derived from the loader application's identity, which consists of its code and layout in memory, and the unique identifier of the application itself which consists of the name and the current version of that application. For en- and decryption, we use authenticated encryption, more specifically AES-128 in CCM mode of operation. The key is derived using an HMAC construction based on Spongent-128.

Our approach maintains the confidentiality of code and data based on a zero-software TCB with two different mechanisms. Firstly, before loading an encrypted application, the code resides encrypted within memory such that no other application is able to read it. Secondly, after an encrypted application has been loaded, the program-counter-based memory-access logic ensures that no other module can access code or data of the decrypted application.

We now illustrate the loading process of an encrypted application:

1. The loader application is started and typically has a code and a data section.

2. It first derives the decryption key from its own identity and the unique identifier of the encrypted application it is about to decrypt.

3. The application is decrypted and checked for integrity simultaneously by using authenticated decryption with the derived key. If the integrity property is violated, all intermediate data is wiped and the loading process is aborted.

4. The program-counter-based memory-access control is activated for the just-decrypted application.

5. The loader application is finished and is now able to load the next encrypted application.

Please note that steps (2), (3) and (4) need to be performed atomically, i.e., interrupts have to be disabled. Otherwise an attacker could read confidential code which has not been protected by the memory-access logic yet.

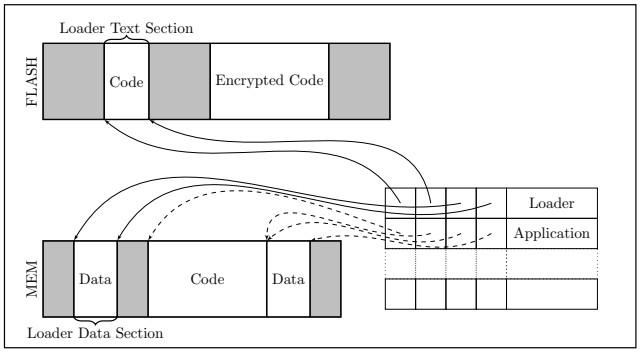


Figure 5: Situation after Loading: Application is decrypted and protected.

Figure 5 shows the configuration of the memory-access logic's boundary registers after the loading process has been completed. The code and data of the loader and the decrypted application are protected. Table 3 shows the different access rights for the whole address space. The only part of an application that remains readable and executable is the entry point. All other areas of an application, i.e., the text section and the data section, are not readable or writeable from any other application or unprotected code, which is ensured by the memory-access logic.

From/To	Entry	Text	Data	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/ Other App	r-x	---	---	rwx

Table 3: Access rights from/to an application enforced by the memory-access logic.

After explaining our architecture, we briefly describe its security properties:

- *Isolation*: Every application is completely isolated from other applications regardless of their privilege level. No other application or unprotected code can read from or write to the code and data section of a given application.
- *Confidentiality*: Confidentiality of code and data for encrypted applications can be guaranteed at any given point in time, i.e., before applications are loaded as well as afterwards. Furthermore, confidentiality can be ensured *offline* due to mutual integrity checks between the loader and the application that is about to be decrypted.
- *Integrity*: We are able to guarantee the integrity of encrypted applications *offline*, meaning that manipulations are already detected at load time rather than communication time.

The confidentiality property is guaranteed by two mechanisms. Before load time, applications are encrypted and thus considered confidential. After load time, applications are protected by the program-counter-based memory access logic and are therefore considered confidential as well. The only possibility for attackers to violate the confidentiality is by compromising the loading process, which is prevented by our design as follows: If the loading position or protected sections of the loader are tampered with when an application is about to be loaded, the decryption key is derived incorrectly because the identity of the loader has changed. The authenticated decryption then fails and the loading process immediately aborts. If the encrypted application is tampered with before loading, the authenticated decryption fails as well and loading aborts again. Tampering is not possible while the loading takes place because authenticated decryption and protecting the application are performed atomically.

5.1.2 Data Isolation through Encryption

Providing traditional isolation mechanisms in modern heterogeneous multi-core architectures ultimately does not scale. On the one hand, approaches based on an MMU (Section 4) need synchronized TLBs for each core to ensure a consistent view of the system's memory. TLB shootdowns negatively impact the overall performance of large systems and therefore new solutions not based on an MMU are required. On the other hand, the boundary registers for the memory-access logic presented in Section 5.1.1 need to be allocated at synthesis time, limiting the system's flexibility. They also need to be allocated in hardware, increasing cost when a large number of application need to be supported.

We present a hardware-based data-isolation mechanism which does not suffer from a scalability problem when used in an invasive-computing scenario, and is also resilient to system-level attackers. It provides data confidentiality through transparent hardware-supported memory encryption. This also makes the architecture compatible with environments that have complex memory hierarchies. Furthermore, it enables the use of shared memory as a lightweight and easy-to-use secure communication channel.

Our solution relies on encryption to ensure the confidentiality of data of different applications sharing the same address space. Since there are no access-control mechanisms, all applications can read from or write to any given address. However, the encryption binds the ciphertext to a specific application, as a unique Initialization Vector (IV) is used for each application. This IV is the concatenation of the application's identifier with the memory address that is about to be read or written. Although a malicious application has the ability to read any location, it will be unable to recover the correct plaintext when trying to access confidential data. However, since our architecture provides confidentiality, but not integrity, and because an attacker can write to any memory location, they can easily corrupt another application's confidential data. After a single bit flip in a word, decryption will fail when the legitimate application reads the value back. However, if an attacker tried to modify control-flow data, e.g., pointers, the system is likely to crash after a few instructions.

An application is identified by the address of its entry point, and therefore all calls to any secure application first have to pass through it. Consequently, applications need to know each other's location. Furthermore, one application must not be able to relocate itself to the entry point of another secure module, as this would give it access to that module's confidential data. Our architecture satisfies both constraints by creating a static layout of all applications running on a single device and placing code into read-only memory. Since there is no preemption mechanism on invasive architectures because of the run-to-completion semantics, applications can safely be assumed to run cooperatively, i.e., to call each other as required.

The encryption unit, which transparently encrypts and decrypts any data leaving or entering the cache, is inserted between the data cache and the main memory. It is considered to be a black box with the following properties: Firstly, in order to isolate the data of different applications, it is able to identify the application

to which the data that is being written back belongs. Secondly, as one of the design goals is to build a scalable isolation architecture which can be used in multi- and many-core systems, it is stateless. Finally, to support secure shared memory, it is possible to reconfigure the symmetric encryption key to a shared key between communicating applications. Note that data will be stored in clear within the cache and CPU registers. Also remember that it is assumed to be impossible for attackers to read these locations (Section 2). To prevent leakage, our hardware and toolchain respectively take care of flushing caches and clearing all registers when the encryption mode is changed. It is controlled through custom instructions which can be used, for example, to turn encryption on or off, or to configure and use secure shared memory.

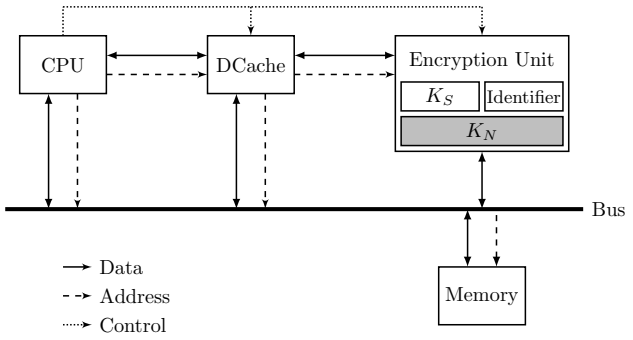


Figure 6: The encryption unit is added between the data cache and memory. When encryption is turned off, the original data signal is sent to the bus; otherwise, the data signal is routed to the encryption unit and the ciphertext is passed to the bus instead. Note that only the control signals for the encryption unit are shown.

Figure 6 shows a CPU's modified core. The encryption unit has three important registers: First, the node key K_N is burned into silicon when the device is initialized, e.g., by blowing fuses. It is the default key which is used to encrypt and decrypt data. Second, K_S can be configured dynamically through a custom instruction and will be used by the secure shared-memory functionality. Third, the identifier register stores the application's identifier, i.e., the address of its entry point.

While building solutions with a minimal hardware TCB is possible, care has to be taken to ensure compatibility with invasive architectures. In particular, heterogeneous memory hierarchies are challenging. Therefore we chose a stateless solution based on transparent data encryption over classical MMU- or boundary-based approaches.

5.2 Isolated Regions in the *i*-NoC

Shared hardware resources which are utilized by different applications are prone to side-channel attacks [1, 16]. This also holds for NoCs, where Wang and Suh [32] demonstrated how a malicious application can extract information from the characteristics of another application's NoC communication. As a counter-measurement, the flows need to be isolated either temporally or spatially. Temporal isolation can be achieved by changing the arbitration scheme, e.g. global Time Division Multiple Access (TDMA). However, finding a feasible TDMA schedule is NP-complete [3, 27] and is not applicable for our envisioned dynamic run-time scenario. More flexible techniques for guaranteed-service connections based on communication budgets allow unused time slots to be used by other applications. Such a technique is adopted in invasive NoCs (*i*-NoCs) [15]. This, however, enables the aforementioned attack (see Figure 8(a)). Therefore, we propose spatial isolation (see Figure 8(b)) implemented by a region-based approach [34].



(a) Side Channel (b) Spatial Isolation

Figure 8: Possible side-channel attack through a shared NoC link [32] and spatial isolation of two applications as a solution to close this side channel.

Each application is mapped to a region of the architecture and utilizes the resources within this region, i.e., computing and networking resources, exclusively. This is incorporated into a hybrid application-mapping methodology [33], as discussed above. A set of compact regions for exclusive usage, called shapes, are explored at design time in a design-space exploration and are mapped during run time without overlaps.

5.2.1 Design-Time Analysis

Starting from a formal graph-based representation for the application (see Figure 7(a)) and the architecture, a design-space exploration creates and analyzes multiple mappings by binding tasks to tiles and routing messages over the NoC (see Figure 7(b)). For each mapping, various and multiple objectives can be evaluated, e.g. throughput, end-to-end latency, or energy consumption. The Pareto-optimal classes of mappings (called *operating points*) are stored and are the input for the run-time management. Constraint graphs can be used as intermediate representation [33]. A constraint graph is a bipartite graph which consists of (a) task clusters, i. e., sets of tasks which are to be mapped onto the same tile, and (b) message clusters, i. e., sets of messages which are routed along the same route. These clusters are accom-

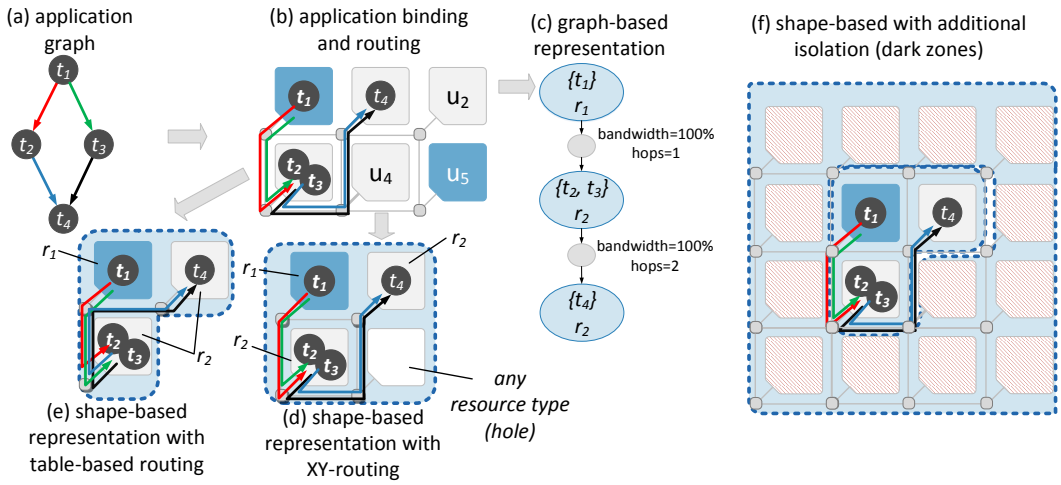


Figure 7: Overview of application-mapping approach (a), (b) and generic representation of an operating point as a constraint graph (c), shapes based on XY-routing (d), table-based routing (e), and extended shapes with dark resources as buffer zones (f).

panied with constraints which need to be satisfied that the run-time mapping adheres to the analyzed mapping at design time. For example, a task cluster must always be mapped exclusively onto a tile. This guarantees the absence of other applications' interference and, hence, side-channel attacks inside a computing tile. However, the constraints for the message clusters only specify a communication budget of time slots that needs to be reserved. No isolation, neither temporal nor spatial, is achieved here unless 100 % of the time slots are allocated (see Figure 7 (c)). This, however, might lead to fragmentation of the NoC as these links cannot be utilized by other messages. To prevent this, we consider isolated mapping regions called *shapes*. These shapes include all allocated tiles executing tasks and routers transmitting data from the application to be protected. It might even occur that a tile is allocated to a shape only because it is connected to a router that is allocated for communication (see Figure 7 (d)). Even more tiles may be added to a shape to enable the buffer zones outlined in Section 4.1 for a even stronger spatial isolation (see Figure 7 (f)).

5.2.2 Run-Time Mapping

The run-time management is responsible for dynamically mapping multiple applications onto the architecture. As detailed before, the design-time analysis results in various shapes that represent regions on the tiled hardware architecture. Each shape represents multiple *shape incarnations*, i. e., regions in the architecture with identical underlying layout of contained resources. The task of run-time mapping is to select one shape incarnation from each active application without any overlap. This is basically a packing problem and can be solved (a) iteratively by sequentially mapping the shape incarnations, or (b) simultaneously by trying to map all active applications concurrently. For the former, the order of mapping the active applications may influence the total number of mapped applications, but fast heuristics can be applied. For the latter, we propose a *SAT-based* mapper which selects the shape incarnations from each application without overlap. We quantified the execution times in experiments with applications stemming from the Embedded System Synthesis Benchmarks Suite (E3S) [11] composed into three different application mixes with five to nine applications. Shape incarnations from these application could be mapped within hundreds of milliseconds with the SAT-based mapper while the heuristics took only in the order of microseconds.[1] However, if there is no feasible solution or there are too many shape incarnations, the SAT-based mapper might take significantly more time. We propose the usage of a time-out to prevent the exhaustive search in case there is no feasible mapping. Additionally, we suggest carefully selecting the number of shape incarnations.

6 Conclusions

Security is an important requirements and it is natural to consider it for invasive applications too. In this paper, we have surveyed different techniques the authors have developed to support security on demand for invasive applications. We have shown that key concepts of invasive computing, namely resource invasion, dynamic resource management, heterogeneous multicore technology, and network-on-chip do not necessarily conflict with security. It is, however, necessary to consider security *during design* of invasive systems and not as a patch later on, and

[1] In contrast to SAT-based mapper, the heuristics were not always able to map all applications.

that security is treated equally at all architectural design layers.

Since techniques finally exist to provide (basic) confidentiality C, we are now aiming to evaluate techniques providing ϵC but experimenting with concrete attacks on invasive systems.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In Peng Ning and Vijay Atluri, editors, *Proceedings of the 2007 ACM workshop on Computer Security Architecture, CSAW 2007, Fairfax, VA, USA, November 2, 2007*, pages 11–18. ACM, 2007.

[2] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[3] Benny Akesson, Anna Minaeva, Premysl Sucha, Andrew Nelson, and Zdenek Hanzálek. An efficient configuration methodology for time-division multiplexed single resources. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 161–171. IEEE Computer Society, 2015.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[6] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: A software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 113–122, New York, NY, USA, 1989. ACM.

[7] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. An X10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112`.

[8] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Dynamic X10: Resource-aware programming for higher efficiency. Technical Report 8, Karlsruhe Institute of Technology, 2014. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000041061`. X10 '14.

[9] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick Drew McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 132–145. ACM, 2004.

[10] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. Invasive computing in HPC with X10. In Michael Hind and David Grove, editors, *Proceedings of the third ACM SIGPLAN X10 Workshop, X10 2013, Seattle, Washington, USA, June 20, 2013*, pages 12–19. ACM, 2013.

[11] Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. `http://ziyang.eecs.umich.edu/dickrp/e3s/`.

[12] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[13] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC'15)*, pages 241–250. ACM, 2015.

[14] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler co-design approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133:1–133:29, 2014.

[15] Jan Heisswolf, Ralf König, Martin Kupper, and Jürgen Becker. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 39(8):2603–2622, 2013.

[16] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[17] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 185–199. IEEE Computer Society, 2000.

[18] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In Ruby B. Lee and Weidong Shi, editors, *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 10. ACM, 2013.

[19] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. Cutting out the middleman: OS-level support for X10 activities. In *Proceedings of the fifth ACM SIGPLAN X10 Workshop*, X10 '15, pages 13–18, New York, NY, USA, 2015. ACM.

[20] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In Ross McIlroy, Joe Sventek, Tim Harris, and Timothy Roscoe, editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)*, volume USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pages 9–14. EuroSys, 2011.

[21] Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. OctoPOS: A hardware-assisted OS for many-cores, 2014. Poster.

[22] Santiago Pagani, Lars Bauer, Qingqing Chen, Elisabeth Glocker, Frank Hannig, Andreas Herkersdorf, Heba Khdr, Anuj Pathania, Ulf Schlichtmann, Doris Schmitt-Landsiedel, Mark Sagi, Éricles Sousa, Philipp Wagner, Volker Wenzel, Thomas Wild, and Jörg Henkel. Dark silicon management: An integrated and coordinated cross-layer approach. *it - Information Technology*, 201X.

[23] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Software Eng.*, 5(2):128–138, 1979.

[24] Johny Paul, Walter Stechele, Manfred Kröhnert, Tamim Asfour, and Rüdiger Dillmann. Invasive computing for robotic vision. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference, ASP-DAC 2012, Sydney, Australia, January 30 - February 2, 2012*, pages 207–212. IEEE, 2012.

[25] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[26] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[27] Faisal Shad, Terence D. Todd, Vytas Kezys, and John Litva. Dynamic slot allocation (DSA) in indoor SDMA/TDMA using smart antenna basestation. *IEEE/ACM Trans. Netw.*, 9(1):69–81, 2001.

[28] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 56(6):280–287, 2014.

[29] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In Sushil Jajodia and Jianying Zhou, editors, *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 344–361. Springer, 2010.

[30] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 2003 International Conference on Supercomputing (ICS-03)*, pages 160–171, New York, June 23–26 2003. ACM Press.

[31] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.

[32] Yao Wang and G. Edward Suh. Efficient timing channel protection for on-chip networks. In *2012 Sixth IEEE/ACM International Symposium on Networks-on-Chip (NoCS), Copenhagen, Denmark, 9-11 May, 2012*, pages 142–151. IEEE Computer Society, 2012.

[33] Andreas Weichslgartner, Deepak Gangadharan, Stefan Wildermann, Michael Glaß, and Jürgen Teich. DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In Radu Marculescu and Gabriela Nicolescu, editors, *2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12-17, 2014*, pages 34:1–34:10. ACM, 2014.

[34] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix Freiling, Michael Glaß, and Jürgen Teich. Design-time/run-time mapping of

security-critical applications in heterogeneous mpsocs. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 153–162, New York, NY, USA, 2016. ACM.