Pluggable Components All The Way Down

Nils Asmussen
nils.asmussen@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Michael Roitzsch
Carsten Weinhold
{michael.roitzsch,carsten.weinhold}@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

ABSTRACT

The hardware and software requirements for the networked computers built into cyber-physical systems are as diverse as the environments in which they are expected to operate. We propose an integrated hardware/software co-design for building custom compute hardware from pluggable components and modular system software that is specifically tailored to these devices.

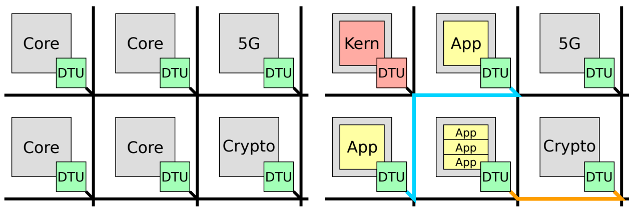
CCS CONCEPTS

• **Computer systems organization → Architectures**; *Real-time operating systems*; • **Security and privacy → Systems security**; *Security in hardware.*

KEYWORDS

tiled architectures, operating systems, security, real-time, low-latency, isolation, co-design, composable

ACM Reference Format:
Nils Asmussen, Michael Roitzsch, and Carsten Weinhold. 2019. Pluggable Components All The Way Down. In *NGOSCPS 2019: 1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems, April 15 2019, Montreal, Canada.* ACM, New York, NY, USA, 3 pages.



(a) Hardware architecture **(b) Software architecture**

Figure 1: Overview of the M^3 architecture

1 INTRODUCTION

Cyber-physical systems (CPSes) and the "Internet of Things (IoT)" are already pervasive in industrial production and they are expected to become ubiquitous in many other sectors, too. For example, connected CPSes have great potential to better automate and optimize critical infrastructure such as electrical grids and transportation networks, but there are also use cases in health care. However, a one-size-fits-all solution for the compute hardware and system software of all these devices is infeasible, primarily due to cost pressure and energy constraints, but also because each domain requires different compute capacity, sensors, and actuators. Instead, customized solutions are needed for both the hardware and the software that drives it. System designers should be able to easily assemble these specialized computers and their operating systems (OSes) from reusable building blocks.

At the software level, microkernels like L4 [3] provide a common substrate to build custom OSes that can be tailored to specific use cases by provisioning only those system services that are needed for the application scenario. Furthermore, all system services, including device drivers, run as isolated user-space programs. The microkernel approach to building system software enables component isolation in a way that helps enforce the principle of least authority. Systems based on this principle improve security by design, as they limit privileges of individual components. Each component also has a smaller attack surface and is isolated by default from failures in other parts of the system.

We argue that microkernel-based OSes are well-suited for CPS and IoT devices, as the underlying construction principle encourages secure-by-design system software. In these connected devices, security is a precondition for safety, as these systems have the potential to cause physical damage to infrastructure or even bodily harm to people. But we believe that microkernel-like ideas can also improve hardware design. Hardware engineers often need to incorporate third-party components into their systems, much like software developers use third-party libraries. For example, system-on-chip (SoC) designers regularly integrate IP blocks from various sources, but not all of them may be trustworthy. A prime example are off-the-shelf wireless modems, which run firmware that implements inherently complex communication protocols, and which may therefore be susceptible to attacks. Similarly to how a microkernel isolates mutually distrusting software components, a small and trusted hardware component could limit the damage that a faulty or compromised hardware block can do to the rest of the system. If carefully designed and integrated with the network on chip (NoC), this hardware component may also allow for simpler integration of third-party IP blocks.

In the following section, we summarize the state of the art of componentized architectures at both the hardware and OS level. Based on that, we describe how pluggable components on both levels can be combined to build customized and secure CPS or IoT platforms. Finally, we discuss the open research challenges.

2 STATE OF THE ART

2.1 Hardware Components

At the hardware level, tiled architectures [10] are already used today. They enable modular system design and an easy integration of different components into one system, but do not solve the problem of isolating potentially untrusted components from each other. M^3 [1] provides a solution by introducing a simple and trusted hardware component into each tile, called data transfer unit (DTU), as depicted in Figure 1a. The DTU serves both the purpose to add a uniform interface to each tile and to isolate tiles from each other. The uniform interface enables the separate development of each component (cores, modems, accelerators, . . .) and simplifies the tile management as well as their collaboration at runtime. M^3 is a hardware/software co-design, where the M^3 microkernel runs on a dedicated *kernel tile*, as shown in Figure 1b. Applications, accelerators, and other IP blocks are referred to as *user tiles*. To establish communication channels dynamically at runtime, the DTU offers a set of endpoints that need to be connected to form communication channels. These channels can only be established by the kernel tile, but not by user tiles. Since M^3 does not impose any architectural requirements (such as different privilege levels or a memory management unit) on the components in user tiles, any component can be integrated. This idea can also be taken further by integrating memory and I/O devices such as disks or network interface cards as user tiles. For example, LegoOS [9] suggested a similar architecture for data centers to increase modularity and improve utilization.

2.2 Software Components

Combining independently developed components to create a larger system is also a standard technique in software engineering. As a system architecture paradigm, microkernels like L4 offer a compelling way to place components into separate address spaces and enable carefully controlled interaction between them. At the same time, microkernels maintain a low resource footprint and provide efficient communication primitives. We believe that these properties are vital for CPSes and that strong isolation between components simplifies reasoning about the overall system. We substantiate this belief with a discussion of existing work using microkernels:

Security. Microkernels address security by reducing the Trusted Computing Base (TCB). The TCB comprises all components that are relied upon for maintaining specific security objectives like confidentiality or integrity. The Nizza security architecture [2] implements a secure VPN gateway with three components on top of a microkernel: one component provides TCP/IP networking for the Internet-facing network adapter, a second component does the same for the intranet-facing network adapter, and a third component implements the VPN cryptography layer between the intranet and the Internet. Regarding confidentiality and integrity of the VPN traffic, only the latter two components have access to plaintext and are therefore part of the TCB. Attacks from the Internet on the VPN device would be absorbed by the Internet-facing component, which never has access to plaintext within the confines of its address space. Subverting components outside of address space boundaries would require a vulnerability within the microkernel or misconfigured communication channels. The former can be addressed with formally verified kernels [5], the latter with communication controlled by capabilities.

Real-Time. Microkernels have been researched extensively for real-time use cases [4]. Many embedded real-time executives and time-partitioning kernels bear close resemblance to microkernel designs. On traditional systems like Linux, many activities happen implicitly within the kernel as background work. On a microkernel however, all activities happen explicitly within components. The system itself does not perform any surprise activities that would perturb scheduling. This principle allows to realize time as a first-class abstraction in microkernel systems [8]. Furthermore, resource managers and I/O schedulers to govern timely access to devices like sensors or actuators can be implemented as components on top of the microkernel rather than being baked into the monolithic kernel of traditional systems. This way, these managers enjoy the same security, fault tolerance, and scheduling advantages as any other component on top of the microkernel.

3 CROSS-CUTTING CONCERNS

After discussing the existing solutions and respective advantages of pluggable components at the hardware and software level, we now bring these two worlds together. We focus on security and real-time, because of their cross-cutting nature, which poses interesting challenges for the hardware/software co-design of future CPSes.

3.1 Security

Physical Separation. More performance-critical CPSes such as autonomous cars or Industry 4.0 machinery require modern general-purpose cores that perform out-of-order execution and speculative execution to deliver the expected performance. Unfortunately, these features and their interactions are complex, which is one of the reasons why security issues like Meltdown, Spectre, and Foreshadow have been lurking unnoticed in CPUs for many years. These security issues allow attackers to leak information, which is in particular a problem for devices that process sensitive data such as patient records. While there are mitigations for the mentioned security issues, the mitigations are similarly complex as the CPU features that introduced them and it is therefore expected that more security problems will be found in the future – either in the CPU or the mitigations. For these reasons, we believe that physical separation of security-critical components is important to prevent such side-channel attacks by design.

Placing software components on different physical cores is supported by most modern operating systems. M^3 runs the kernel on a dedicated tile and thereby also prevents side channels between the kernel and applications by design. However, resource constraints will prevent us from using dedicated cores for all applications. In other words, some applications can be placed horizontally on different cores, whereas some applications need to be stacked vertically on a single core (see Figure 1b). This trade-off raises several research challenges. First, how and when do we decide which applications run on dedicated cores? Second, how can M^3's model of DTU-based communication between tiles be combined with core-local inter-process communication in case of vertical stacking? And third, what are the latency differences between these communication types?

Software Components. We have already seen some advantages of microkernel-based operating systems in the work mentioned in subsection 2.2. Another important point is the ability to update individual components. For example, many devices still run old Linux kernels, because the device manufacturer needed to develop custom device drivers, which are part of the kernel (as a kernel module or builtin) in Linux' model. Since Linux' internal API is a huge moving target, vendors need to keep up with Linux' development speed in order to keep the system secure. In practice, many systems remain unpatched. We believe that microkernel-based OSes are better suited to address this problem. However, to enable updates and restarts of individual components without affecting the rest of the system, standardized interfaces between the components are required, which is still an open challenge.

Another problem is that reality forces us to reuse and combine existing components. Ideally, we would like to freely choose the thickness of isolation walls between such components. In some cases, we need maximum performance and are willing to sacrifice security to some degree. In other cases, we prefer security over performance and therefore want to place the components in different address spaces or on different cores. Unfortunately, there is still tooling missing to support this trade-off with acceptable effort. One promising example is Sandcrust [7] that allows Rust applications to use an existing and untrusted library by running it in a separate address space. At the same time, this only requires a simple annotation of the function to sandbox, making it easy to use.

Remote Attestation. Cooperating software components need to trust each other for certain security goals. Within a device, the microkernel-based OS can establish this trust implicitly by launching software components and configuring their communication channels. However, for use cases that offload part of the processing to a data center, the trust relationship spans multiple devices. Remote attestation techniques can securely identify remote components using hardware-based trust anchors and cryptographic protocols. An interesting research question is how we can minimize the hardware requirements using hardware/software co-design in our M^3 architecture.

3.2 Real-Time

As another topic that needs to be respected on all layers, we regard real-time as a combination of predictability in timing behavior and low-latency dispatching of activities.

Distributed Compute Resources. Predictability can be catered by integrating an existing real-time application onto a dedicated core for real-time tasks. This core could run a traditional real-time operating system and would constitute the real-time island of the system, while other hardware tiles could host non-real-time workloads. This combination allows for maximum reuse of existing components, but enables only limited interaction of real-time workloads across hardware resources.

Interesting combinations of new hardware components, like computer vision accelerators with real-time software components, require a much deeper integration of timing behavior across different compute resources. M^3 would enable such heterogeneous resources to interact directly, but timing models and schedulability analysis for such systems remain challenging. These connected compute resources communicate with each other to solve an overall problem, but they execute concurrently and independently, essentially forming a distributed real-time system. Research on systems with both CPUs and GPUs, for example, is still ongoing.

Low-Latency Communication. Timely interaction between components requires low-latency communication channels. Microkernels solve this problem for single-core systems with fast kernel entry and exit paths, but cross-core communication remains a performance problem. Decoupling [6] allows a multicore system to be co-scheduled by both a traditional Linux system and a microkernel to combine the support for existing software with the predictable low-latency scheduler response of the microkernel.

If applications span across multiple separate cores, for example to improve security by removing cache side channels, M^3 in combination with the DTUs can offer direct fast-path communication between the cores. But latency guarantees for the on-chip interconnect remain an open problem, with priority channels or credit systems as potential solution ideas.

4 CONCLUSION

We outlined how componentized, microkernel-based OSes can be integrated with tile-based hardware architectures and chip-level communication control in order to build customized CPS and IoT devices. The isolation-by-default approach that is central to M^3 encourages secure-by-design systems at both the hardware and the software level. But important research challenges remain. For the hardware part, the trade-off between physical separation to improve security and sharing of resources to reduce cost depends on the specific use case. With regard to software, it is still difficult to decide where to place isolation boundaries. Finally, real-time analysis for the inherent parallelism in tile-based systems is challenging. Better tool support is needed to help system designers in all these areas.

5 ACKNOWLEDGEMENTS

This research was co-financed by public funding of the state of Saxony/Germany.

REFERENCES

[1] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 189–203. ACM, 2016.

[2] Hermann Hartig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The nizza secure-system architecture. In *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 10–pp. IEEE, 2005.

[3] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In *SOSP*, volume 97, pages 66–77, 1997.

[4] Hermann Härtig and Michael Roitzsch. Ten years of research on l4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

[5] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[6] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Decoupled: low-effort noise-free execution on commodity systems. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2016.

[7] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 51–57. ACM, 2017.

[8] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In *Proceedings of the 13th EuroSys Conference*, page 26. ACM, 2018.

[9] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18. USENIX Association, 2018.

[10] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 10 2007.