Component-based OS Design for Dependable Cyber-Physical Systems

Gabriel Parmer, Runyu Pan, Yuxin Ren, Phani Kishore Gadepalli, Wenyuan Shao
The George Washington University
{gparmer,panrunyu,ryx,phanikishoreg,shaowy}@gwu

ABSTRACT

Cyber-Physical Systems (CPSes) require the difficult combination of both high levels of code assurance *and* significant software complexity. Conventional software systems are designed either for high assurance – RTOSes and the minimal software of traditional embedded systems, *or* feature-rich complexity – POSIX and general purpose programming environments with package management systems to control complexity. Our ability to consolidate these conflicting demands will impact the security, reliability, and predictability of the systems that tightly control and expand the capabilities of our physical world.

Though many systems have focused on completely isolating the most critical functionality from that which is feature-rich, we propose a more nuanced approach to enabling high-assurance. Functionality and isolation properties are explicitly *designed* via component selection and dependency resolution. A shared set of components enables strong code sharing and hardening, thus bolstering system assurance. To accomplish this, a system foundation is required that is *compositional* across components with respect to the non-functional concerns of security, reliability, predictability, and scalability. We outline the requirements for such a system, and discuss an existence proof of many of the necessary features.

1 BACKGROUND AND MOTIVATION

Existing Operating System (OS) designs have taken us very far. Linux is the de-facto OS in almost every application domain, with the exception of deeply embedded systems. However, software is in the process of crossing an important threshold, from being a binding agent between people and between ideas, to directly controlling most aspects of the physical world. Engineering disciplines that are comparably responsible for the safety of the physical world (for example, civil and mechanical engineering) must adhere to strict standards, certifications, and regulation. Unfortunately, this development methodology is counter to the software development methodologies that have given rise to the most complex software ecosystems, often iconified by Facebook's motto to "move fast and break things".

We posit that a next-generation software infrastructure is required that can address the challenge of marrying software of immense complexity, with trustworthy control of the physical environment, while scaling in resource consumption from one to many cores. First, we'll investigate the risks and shortcomings of current software structures (§1.1 and §1.2), then we'll discuss the necessary properties of future OSes (§2 and §2.1), and the inroads that have already been made (§3).

1.1 Beyond POSIX

The POSIX standard has stood the test of time as an enduring foundation for application development. Whereas previous attempts to "fix" POSIX such as Plan 9 update the system's core underlying abstractions, its constant extension and expansion has demonstrated that the abstractions and mechanisms that underlie POSIX are insufficient for modern systems. Though Linux has served as an adaptable sub-straight for the deployment of most computational tasks, this ad-hoc extension over time has only increased its complexity. From the size of its code base, to its API surface area, to its extensive configuration, the Linux kernel is a difficult and moving target for high-confidence systems. While older Linux deployments are often more trustworthy as they are "battle-tested", they are plagued by a constant stream of bugs and security compromises which are challenging in intermittently connected embedded deployments.

Through the real-time patches, Linux has been adapted to exhibit very low interrupt response times by removing most operations that disable interrupts. However, POSIX and all of the Linux extensions make it difficult to have confidence that every possible execution path exhibits this behavior, partially due to the complexity of the kernel's interactions and large testing surface. Importantly, the focus on interrupt response time ignores unbounded library and kernel paths for most system services that are necessary for high-functionality, multi-processes applications.

Discussion. The economy of making monolithic POSIX systems dependable is challenging: a *single* buggy line of code can lead to a compromise or fault; a *single* bit-flip in the large-footprint data-structures can cause erroneous behavior; a *single* unbounded execution path can cause unpredictable behavior; and a *single* cache-line bouncing between cores can prevent reasonably bounded coordination. In extremely complex code-bases, these factors are almost by definition untestable, thus leading to inevitable failures in systems deployed for decades.

1.2 Beyond RTOSes

In contrast to complex POSIX environments, Real-Time Operating Systems (RTOSes) often provide simple APIs with light hardware abstraction layers for devices, and compile-time specialization to minimize overheads and memory consumption. Traditional RTOSes are static and monolithic: applications and OS are compiled into a single binary, often with no isolation. This design decision is often based on simplicity, and on minimizing the maximum overhead for handling interrupts. However, it complicates system security, update, and flexibility.

Additionally, a lack of effective, built-in multi-core support is a significant challenge for RTOSes. Adapting existing code-bases to parallel execution is challenging, and maintaining both predictable and efficient execution in the face of ever-increasing core counts, is often impossible for conventional lock-based systems.

Discussion. CPSes with significant functionality – thus complexity – require means to mitigate compromises, reduce fault propagation, and efficiently use all available resources, including multiple cores. Traditional RTOSes focus on simplicity at the cost of functionality, which makes them a complicated foundation for feature-rich requirements.

2 COMPONENT-BASED OS DESIGN

For a future OS infrastructure to scale from small microcontrollers, up to multi-core processors, and from simple control loops, to machine-learning, computer vision, and cloud collaboration, we posit that a more flexible organization is required that gracefully adds functionality only when required, and customizes the resource management policies, system abstractions, and isolation boundaries to system requirements. This will enable the code-reuse and isolation necessary to scale in complexity, while specializing system behavior to the specific system requirements.

Package management. Software dependency managers are the de-facto technology to manage the complexity of massive modern software environments. They exist for OSes (`apt`, `brew`, and `rpm`), and for modern languages (`pip`, `npm`, and `cargo`). An application that wishes to use a functionality need only specify it as a dependency, and the package management system resolves the required transitive dependencies.

Standards. Standards serve the roles of enabling applications to program to a specific assumed OS behavior, and of enabling certification of OSes with respect to some standards. Such specifications include the AUTOSAR set of standards for automotives, ARINC 653 (the Avionics Application Standard Software Interface), DoD's Future Airborne Capability Environment (FACE), NASA's core Flight System (cFS), and middle-ware environments such as TAO.

Component-based OS (CBOS) design. CPS OSes require the vibrant eco-system of functionality provided by package management systems, combined with the ability to certify their code against well-defined standards. Component-based system design is focused on *composing* functionality out of *components*. Components are units of code and data that explicitly export a functional, polymorphic interface, and specify a set of interface dependencies. Components are units of potential isolation, thus pairing a system's functional building blocks with memory protection barriers. This partitioning of data-structures, along with the control-flow integrity provided by defined interface entry points, ameliorates many of the challenges with existing monolithic systems.

CPSes require strict non-functional constraints such as timely processing of control loops and planners, and strict memory limits. As system functionality and policies are built up explicitly from components, OSes should scale from microcontrollers to massively multi-core systems. Standards are specified on a subset of the component interfaces, and components are managed like packages.

We identify two requirements to enable the utility of CBOSes in CPSes: (1) *Component-definition of all system policies.* Functional and non-functional properties of the system should be configurable as components. (2) *Non-functional Cross-Cutting Concerns (CCCs).* Non-functional constraints must be maintained as a system is created from constituent components. Of particular interest are CCCs such as *security, predictability, reliability, and scalability.*

2.1 Design for Cross Cutting Concerns

Non-functional Cross-Cutting Concerns (CCCs) are desirable system attributes that are the product of *all* code that is executed by and for an application. As such, *any line of code* can compromise a CCC. CCCs we'll consider include:

- *Predictability.* For an application to exhibit predictable execution, all code executed from interrupt arrival, to I/O output must execute within a bounded, known amount of computation. As such, *all software* that implements the system's abstractions and resource management policies leveraged by an application impact its predictability.
- *Security.* The security of a system depends on both the correctness and exposure to external inputs of each line of system code. The vulnerability of an application, then, is dependent on all code it relies on.
- *Dependability.* Software bugs and hardware errors (*e.g.* due to single-event upsets) can cause system failures that can impact correct CPS control. All code that is relied on by a specific application represent it's surface area of failure risk.
- *Scalability.* The ability to efficiently and predictably use an increasing number of cores is dependent on the scalability of *all* code executed. A single lock, and in some cases, a single cache-line that is shared across cores can significantly harm performance.

Given that a *single line of code* can significantly harm each CCC, system software must be viewed as the Trusted Computing Base (TCB) in secure systems. From this perspective, including only necessary components based on dependencies minimizes this TCB.

3 NEXT-GENERATION FOUNDATIONS FOR CPS OSES

In our view, next-generation OSes for CPSes must fulfill the requirements of component-based design, and enable system design that is optimized to satisfy each of the non-functional CCCs. We identify the core challenges for OS design and implementation to include:

- *Efficient isolation.* Resource partitioning constrains the propagation of faults and compromises. This benefits security and reliability, and generally alters the fundamental economics around the CCCs by constraining the scope of impact of any single line of code. However, isolation isn't free. Memory isolation requires inter-protection domain communication; temporal isolation requires scheduling, context switch, and interrupt overheads; and I/O isolation requires device multiplexing. Reasonable isolation overheads are required for a dependable, CBOS.
- *Policy Customization.* CPSes require both tight control over the non-functional properties of the system, and the ability to have confidence in the correct functioning of the entire software system. Thus the policies and total amount of necessary code must be component-defined and configurable. As modern CPSes require both high confidence and high functionality code to coexist, different policies and abstractions for different applications must coexist.

- *Composability.* Two separate components that provide strong guarantees with respect to a CCC, when attached together via a functional dependency, must *compose* with respect to the CCC. The assurance of an application depends entirely the composition of components it depends on.

To understand the complexities behind these challenges, but also past successes, a few examples:

Inter-Process Communication (IPC). μ-kernels strive to provide efficient isolation by moving many services to user-level "servers" that use IPC for coordination. The detailed mechanisms behind this IPC have a significant impact on the composability of the system. Asynchronous IPC between threads (e.g. POSIX pipes) requires dependency-aware schedulability analysis, and complex buffer-size analysis. In contrast, IPC via synchronous rendezvous between threads has proven to be very fast [4], but has challenging interactions with security [7] and predictable execution [6]. Two predictable components, functionally bound via IPC, might no longer be predictable, or might suffer degraded latency bounds. Alternatively, IPC via thread migration [1] decouples scheduling and execution context, and flows a single scheduling context across components. In this way, timing properties persist across components, thus compose. Even in this case, thread migration requires careful execution context management [11]. Correspondingly, synchronous rendezvous models have been moving toward thread migration [5, 8, 9] to address this challenge.

Kernel synchronization. Inter-component coordination is often mediated by the kernel. Thus the scalability properties of the kernel, and especially of the IPC path, impact the composability of components with respect to scalability: the ability of multiple scalable components to compose into a scalable system is threatened by kernel side-effects spanning from locks, to IPI-based coordination that can cause livelock [3]. In fact, we posit that only a kernel that is entirely wait-free enables the requisite composition with respect to scalability [10].

Timing policy. As the timing properties of the system are integral to the correctness of CPSes, they must be component-defined and configurable. Further, the system must support the co-existence and cooperation between *multiple* mutually untrusting schedulers. Composition of timing properties has been studied in two ways: hierarchical scheduling theory determines if multiple schedulers, when composed, can meet task timing constraints; and temporal capabilities [2] enable schedulers to explicitly coordinate while maintaining temporal invariants.

Kernel simplicity. An open question is to what extent a kernel that enables composability of CCCs, and component-definition of policy can exist without drastically increasing the TCB. There reason for optimism: the Composite kernel's architecture-independent code is less than 6K Lines of Code (LoC). For context, FreeRTOS's (version 10.2) architecture-independent code is over 9.4K LoC.

3.1 Design Principles

What are the principles that underlie the creation of a system that addresses these core challenges? Liedtke [4] provides guidance for μ-kernel design: "...a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality." In this vain, we propose two *additional* design principles for CPS CBOSes.

Design Principle #1. Component-based customization of all system services is fundamental to the design:

> Kernel mechanisms must enable the definition of resource management policies in user-level components, and the additive construction of high-level behaviors from components.

Design Principle #2. The CCCs must be explicitly accommodated in the system's design:

> The kernel should include minimal but strong facilities for component-centric resource isolation, while enabling CCC-constrained composability.

4 CONCLUSIONS AND FUTURE WORK

CPSes require the challenging combination of significant functionality with high assurance, predictable execution. To address this challenge, we propose the use of component-based system development which enables effective code-reuse combined with the composition of components to create complex behaviors with effective isolation. However, the success of such a system requires a CBOS designed for compositional behaviors with respect to predictability, security, dependability, and scalability.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1815690, and by ONR Awards No. N00014-14-1-0386, ONR STTR N00014-15-P-1182 and N68335-17-C-0153. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF nor ONR.

REFERENCES

[1] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition.*
[2] Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. 2017. Temporal Capabilities: Access Control for Time. In *Proceedings of the 38th IEEE Real-Time Systems Symposium.*
[3] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. 2019. Chaos: a System for Criticality-Aware, Multi-core Coordination. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*
[4] J. Liedtke. 1995. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles.* ACM.
[5] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-context Capabilities: A Principled, Light-weight Operating-system Mechanism for Managing Time. In *Proceedings of the Thirteenth EuroSys Conference (Eurosys).*
[6] Sergio Ruocco. 2008. A Real-Time Programmer's Tour of General-Purpose L4 Microkernels. In *EURASIP Journal on Embedded Systems*, Vol. 2008.
[7] Jonathan S. Shapiro. 2003. Vulnerabilities in Synchronous IPC Designs. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy.* IEEE Computer Society, Washington, DC, USA, 251.
[8] Udo Steinberg, Alexander Bottcher, and Bernhard Kauer. 2010. Timeslice Donation in Component-Based Systems. In *OSPERT.*
[9] Udo Steinberg, Jean Wolter, and Hermann Hartig. 2005. Fast Component Interaction for Real-Time Systems. In *ECRTS.*
[10] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. 2015. Speck: A Kernel for Scalable Predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*
[11] Qi Wang, Jiguo Song, and Gabriel Parmer. 2011. Stack management for hard real-time computation in a component-based OS. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS).*