Editor: Mohamed Kaâniche, mohamed.kaaniche@laas.fr

Building Critical Applications Using Microservices

Christof Fetzer | Technische Universität Dresden

Applications are increasingly critical for ensuring business success, mission completion, and even human safety. The standard approach to building critical applications relies on a bottom-up strategy: we must begin with a dependable foundation—the correct CPU and system software—on top of which we build and run our critical applications. However, ensuring the correctness of the CPU and system software (OS, hypervisor, resource management system, and so on) is a grand challenge. In response, Gerwin Klein and his colleagues proposed and demonstrated formal methods to prove the correctness of a microkernel-based system.[1]

Modern systems such as the cloud, embedded systems, and smartphones run Linux, an OS with a monolithic kernel. Safety-critical automotive software is even being run on top of Linux.[2] Proving Linux's correctness, however, appears to be beyond the state of the art of formal proofs: instead of verifying only the 10,000 LOC of a microkernel,[1] one would need to verify the more than 20 million LOC of the Linux kernel. Moreover, there's empirical evidence that Linux always contains bugs that need to be fixed, so a formal proof of correctness would fail anyhow.

Static analysis of open source code repositories indicates approximately 0.61 defects per 1,000 LOC.[3] The analysis of Linux shows that, despite an increasing number of defects being fixed, there are always approximately 5,000 defects waiting to be fixed.[3] Not all of these defects can, however, be exploited for security attacks. Another analysis found that approximately 500 security-relevant bugs were fixed in Linux over the past five years[4]—bugs that had been in the kernel for five years before being discovered and fixed. Commercial code had a slightly higher defect density than open source projects.[3] Hence, we need to expect vulnerabilities in commercial software too.

Application-Oriented Safety and Security

Underlying system software will always contain bugs that attackers can exploit. Given that, we need to build critical applications in a way that doesn't depend on the underlying system software's correctness to ensure the confidentiality and integrity of applications. In the case of business- and safety-critical applications, we must also ensure that the CPU executes the correct program.

Similar to system software, CPUs' complexity has increased dramatically in recent decades. Modern CPUs contain billions of transistors, which makes them susceptible to not only soft errors but also design faults. The use of lockstep CPUs—common in mainframes and embedded CPUs—can detect soft errors but not design faults. High-end CPUs are highly nondeterministic, making it nearly

Copublished by the IEEE Computer and Reliability Societies

impossible to provide lock-step execution without having a major impact on performance and requiring major redesigns.

Rather than considering the OS and other system software trustworthy—as system software designers do—application designers trust only the application components under their control. When running a critical application in a cloud, the system services and, in some cases, the OS would be under a third party's administrative control. New CPU extensions, in particular Intel's Software Guard Extensions (SGX; software.intel.com/en-us/sgx), allow applications to keep their states in encrypted memory (enclaves), thereby preventing even privileged software such as the OS and the hypervisor from accessing the data.

A major limitation of SGX's current implementation is that it slows down by more than an order of magnitude if the working set of an enclave doesn't fit in the SGX extended page cache (EPC), which is currently only about 90 Mbytes. Hence, the state inside the enclave must be kept small—not only to minimize the trusted computing base's size but also to ensure reasonable performance. To achieve this, we have implemented and evaluated a microservice-based approach, as shown in Figure 1.

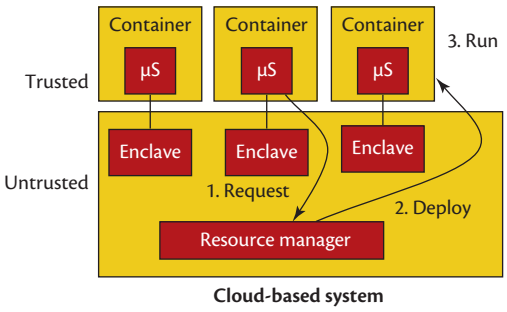

Figure 1. A microservice-based approach to keeping the state inside enclaves small. This not only minimizes the trusted computing base's size but also ensures reasonable performance.

A Microservice-Based Approach

Microkernel-based systems arguably have the better architecture for building trustworthy systems. However, we often build these trustworthy systems on top of monolithic kernels such as Linux. Therefore, my colleagues and I have implemented an approach to building trustworthy systems on top of legacy hardware and software components in which we have limited trust. Such an approach helps us ensure integrity, confidentiality, and correct execution.

In a microservice-based approach, an application is split into a set of microservices. Each microservice focuses on a single aspect. The microservices are loosely coupled and support horizontal elastic scaling; crash and performance failures are tolerated by spawning new service instances to replace crashed or slow instances. Microservices match well with modern container management frameworks such as Kubernetes (kubernetes.io) or Docker Swarm (www.docker.com/products/docker-swarm). These frameworks support high availability, load balancing, and elastic scaling of applications. These frameworks aren't yet available in the embedded domain or in automotive systems but are expected to be soon.

I briefly describe how we ensure the integrity, confidentiality, and correct execution inside microservices with the help of both software and CPU extensions (the technical details and performance evaluations can be found elsewhere[5–8]).

Ensuring Correct Execution

Depending on an application's criticality (say, at most one system failure in 10^9 operating hours is permitted), you can't always assume that the CPU is correctly executing the application. Modern CPUs will detect and mask most soft errors. However, the detection rate is insufficient for critical applications. Because high-performance CPUs are highly nondeterministic, a traditional lock-step execution isn't possible. Instead, a software lock-step execution should be performed. The benefit of software-based protection is that it can be combined with a fast recovery mechanism using the hardware transactional memory of modern CPUs; we've shown that doing so not only detects but also masks most soft errors. The resource overhead (about 100 percent) is similar to hardware lock-stepping but can tolerate about 90 percent of the soft errors and support nondeterministic CPUs and applications.[5]

Some critical applications' requirements can be so high that you need to detect CPU design faults too. For these cases, applications can be transformed such that computations happen in a homomorphic space in which wrong executions by the CPU can be detected:[6] because of the program space's pseudorandomization, a wrong result will most likely not be a member of the homographic space. This homomorphic execution is performed only for safety and not security, so the overheads stay within about 150 percent, which is close to the resource overhead of hardware lock-step execution,

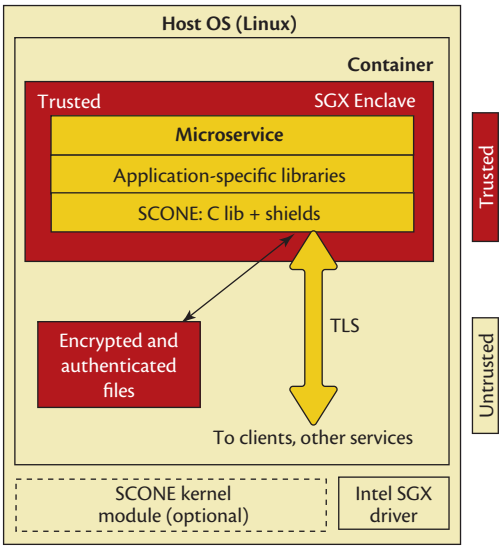

Figure 2. Each secure container runs a single microservice instance inside an Intel Software Guard Extensions (SGX) enclave. This ensures confidentiality and integrity even against attacks from software with higher privilege such as the OS or hypervisor.

especially when you consider that lock-step cores must be slowed down to remain in sync despite, say, error code corrections by one core.

Integrity and Confidentiality

To ensure the confidentiality and integrity of microservices, we've implemented the abstraction of a secure container.[7] The container engine—in our case, a plain Docker engine—can't distinguish a secure container from a regular container. Each secure container runs a single microservice instance inside SGX enclaves (see Figure 2). The CPU registers, the main memory, the files, and the network communication are transparently encrypted such that we can ensure confidentiality and integrity even against attacks from software with higher privilege such as the OS or hypervisor. Throughput is close to native speed—as long as the enclaves fit inside the EPC.

Protection against attacks via the system call interface (such as Iago attacks) is simplified with microservices. Microservices often replace OS services (such as a file system service) with another microservice to ensure better scalability and avoid residual dependencies on individual hosts. Hence, many microservices use only a small set of system calls that can be transparently protected by the secure container infrastructure.

Microservice-based applications consist of new and existing microservices. For example, you might use services like memcached and Redis as part of an application. Many of those existing services will use unsafe languages like C and C++. Similar to protecting the OS kernel,[4] we need to protect such services against common attacks via their network API, such as buffer overflows and format string attacks. Hence, we implemented a compiler-based approach that provides additional protection for existing code.[8] Because critical applications must be available, this approach tolerates faults, such as out-of-bound accesses, and stops a service only if it's unsafe to continue.

Microservices combined with secure containers can facilitate new ways of building critical applications. These applications will benefit from tools and services built for less critical software. Secure containers and compiler extensions can help address the more stringent requirements of critical applications.

Although this approach is sufficient for implementing fail-stop applications, there are still several open research questions regarding whether and how it might support fail-operational applications. ■

Acknowledgments

The EU Horizon 2020 projects Secure Enclaves for Reactive Cloud Applications (645011) and SecureCloud (690111) and the Center for Advancing Electronics Dresden partially supported this work.

References

1. G. Klein, G. Heiser, and T. Murray, "It's Time for Trustworthy Systems," *IEEE Security & Privacy*, vol. 10, no. 2, 2012, pp. 67–70.
2. L. Bulwahn, T. Ochs, and D. Wagner, *Research on an Open-Source Software Platform for Autonomous Driving Systems*, tech. report, BMW Car IT GmbH, Oct. 2013; www.bmw-carit.de/downloads/publications/ResearchOnAnOpenSourceSoftwarePlatformForAutonomousDrivingSystems.pdf.
3. "Coverity Scan Open Source Report 2014," Coverity Scan, 2015; go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf.
4. K. Cook, "The State of Kernel Self Protection Project," Linux.com, 12 Sept. 2016; www.linux.com/videos/state-kernel-self-protection-project-kees-cook-google.

5. D. Kuvaiskii et al., "HAFT: Hardware-Assisted Fault Tolerance," *Proc. 11th European Conf. Computer Systems* (Eurosys 16), 2016, article 25.
6. D. Kuvaiskii and C. Fetzer, "Δ-Encoding: Practical Encoded Processing," *Proc. 45th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 15), 2015, pp. 13–24.
7. S. Arnautov et al., "SCONE: Secure Linux Containers with Intel SGX," *12th USENIX Symp. Operating Systems Design and Implementation* (OSDI 16), 2016; www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.
8. M. Brünink, M. Süßkraut, and C. Fetzer, "Boundless Memory Allocations for Memory Safety and High Availability," *Proc. 41st Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 11), 2011, pp. 13–24.

Christof Fetzer is a professor of computer science at Technische Universität Dresden. Contact him at christof.fetzer@tu-dresden.de.