



Open Kernel LabsTM

Be open. Be safe.

TECHNOLOGY WHITE PAPER

**VIRTUALIZATION AND
COMPONENTIZATION
IN EMBEDDED SYSTEMS**

**AND HOW IT WILL CHANGE
THE WAY YOU ENGINEER**

**Josh Matthews
Field Application Engineer
Open Kernel Labs, Inc.**

October 2008

INTRODUCTION

Continually increasing complexity in mobile and consumer electronics design, coupled with intense market pressure, make applying the same-old approach to embedded development a competitive liability for mobile and consumer electronics developers.

Implementing next-generation software architecture based on principles of componentization and virtualization provides a competitive advantage to developers that are willing to change.

We discuss how a microkernel-based hypervisor providing virtualization and componentization supports an evolution to a more fault resilient, secure, and easily maintained software architecture and implementation.

IT'S GETTING HOT IN HERE

THE RAPID PACE OF CHANGE IN THE MOBILE AND CE MARKETPLACE

The mobile and consumer electronics marketplace is experiencing a revolutionary transformation from its traditional focus on communications, mechanical and electrical engineering to an emphasis on adding value through software functionality and the provision of multimedia content and high value services.

These changes have heralded a massive growth in the market. With this growth has come a significant shift in consumer expectations. In a short decade we have witnessed the move from an industry that provided simple, static, black box devices, to an environment in which the consumer demands that the devices in their pockets, on their shelves, and in the cloud be nothing less than a supporting limb of their digital life.

This transition does not depart from traditional needs for quality and a reliable "instant-on" end-user experience but adds new requirements for security and robustness in an increasingly connected world; the resulting intense market pressure means only one thing: we must change the way we engineer, and we must engineer the change.

But how can we change? We first must recognize the underlying trends and requirements that result from these significant market shifts.

The move to user-customizable and open source application environments

The consumer now demands more from their mobile and CE device than the software it ships with. With the dramatically increasing ubiquity of the Internet; the associated increase in content creators and consumers that is the hallmark of Web 2.0; the rise of open and free operating systems such as Linux; and the desire for developers to participate in a growing mobile application ecosystem, the consumer expectation is clear: it's their device and they want to control it. This creates several key issues for device manufacturers and application engineers: how to relinquish control while still guaranteeing a consistent and reliable user experience, all the while dealing with the potential for conflicting licensing issues and the need to maintain a competitive advantage.

The need to create embedded designs with greater security

Consumers are now doing more with their device than ever before: initiating financial transactions, exchanging private emails, storing sensitive data, participating in enterprise networks. This presents a challenge to device engineers, particularly with the move to user customization, of how to provision a device with the highest possible level of security that will earn consumer trust in an increasingly untrustworthy environment. The primary consideration here is the size of the device's Trusted Computing Base (TCB) – the part of the system that can circumvent security. This demand can only be met via a dramatic minimization of the TCB. This is a demand that is increasingly at odds with the rapid rise in the complexity of the device.

The demand for high reliability

The increasing positioning of the device as the cornerstone of consumer's digital lives places an unprecedented demand on the reliability of that device. Consumers expect nothing less than an always-on experience. Hard reboots are no longer an acceptable substitute: the device must take care of itself, even in the face of uncontrollable application environments.

Responsiveness from go to "whoa"

It is obvious consumers want to do more, and a key challenge for developers is that they demand to do it *faster*. With the recognition that the device is an extension of, and increasingly a replacement for, their personal computing platform, the user has an accompanying expectation that their application experience will be the same. With increasing demands on device size and power management, the requirement is clear: we must do more with less, and we must do it seamlessly.

Increasing competitiveness in platforms

OpenMoko, LiMo, Android – the industry is in a state of flux. 2008 heralds a new dawn in the development of mobile platforms as one way of addressing consumer need for open application environments. The problem for device manufacturers and application developers is uncertainty: it is unclear which platform will be the consumer choice. This is just the beginning.

The need to support legacy migration

The rapidly changing marketplace doesn't negate the massive investment manufacturers have placed in existing system software. Base-band stacks, legacy RTOS's, and existing application infrastructure are still useful, relevant, and need to be supported. The issue is they will be executing in an unforeseen harsh environment alongside open application platforms. These modules must be deployed, isolated, and protected against potential attacks from the rest of the system.

A reduction in time to market expectations

The phenomenal explosion in consumer demand has placed a key requirement on maintaining a competitive advantage - release better devices, with more capability, and do it in less time. Rapid prototyping, rapid application deployment, legacy migration, and the ability to develop and support a product series quickly are all demands resulting from this requirement. We must simply engineer faster than ever before.

Hardware consolidation and BOM cost minimization

Reductions in device size, coupled with market conditions that are increasingly dictated by low prices, present a clear need to minimize BOM cost as dramatically as possible whilst maintaining functionality. The solution is just as clear: squeeze more out of the same hardware, consolidating competing system software that traditional engineering would have decoupled. This presents unique challenges in guaranteeing the security and reliability of the device.

ENGINEERING CHANGE VIA VIRTUALIZATION AND COMPONENTIZATION

Engineering the change required in the mobile and consumer electronics industries in response to the significant market pressures and its accompanying requirements is no small task. Nor should it be. You are being asked to develop systems that are user-customizable, support open, free, and uncontrollable application environments on potentially several competing platforms, that are more secure, reliable, and responsive than ever before, and you need to do it all with rapidly increasing expectations on time to market and cost.

The solution is modeling your underlying system on a platform that can support the complementary features of virtualization and componentization.

What is it?

Virtualization refers to providing a software environment on which programs, including operating systems, can run as if on bare hardware (Figure 2.1). Such a software environment is called a virtual machine (VM). A VM is an efficient, isolated duplicate of the real machine.

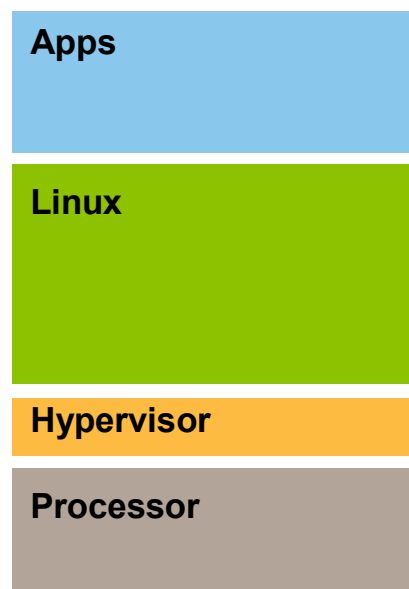


Figure 2.1.

A virtual machine. The hypervisor (or virtual-machine monitor) presents an interface that looks like hardware to the “guest” operating system.

The software layer that provides the VM environment is called the *virtual-machine monitor* (VMM), or *hypervisor*.

In order to maintain the illusion that is incorporated in a virtual machine, the VMM has three essential characteristics:

1. the VMM provides to software an environment that is essentially identical with the original machine;
2. programs run in this environment show, at worst, minor decreases in speed;
3. the VMM is in complete control of system resources.

All three characteristics are important, and contribute to making virtualization highly useful in practice. The first (similarity) ensures that software that runs on the real machine will run on the virtual machine and vice versa. The second (efficiency) ensures that virtualization is practicable from the performance point of view. The third (resource control) ensures that software cannot break out of the VM.

Componentization is a complementary concept to virtualization. Componentization entails the ability to segment and distribute large and complex software into a number of simpler, isolated components with smaller trusted computing bases.

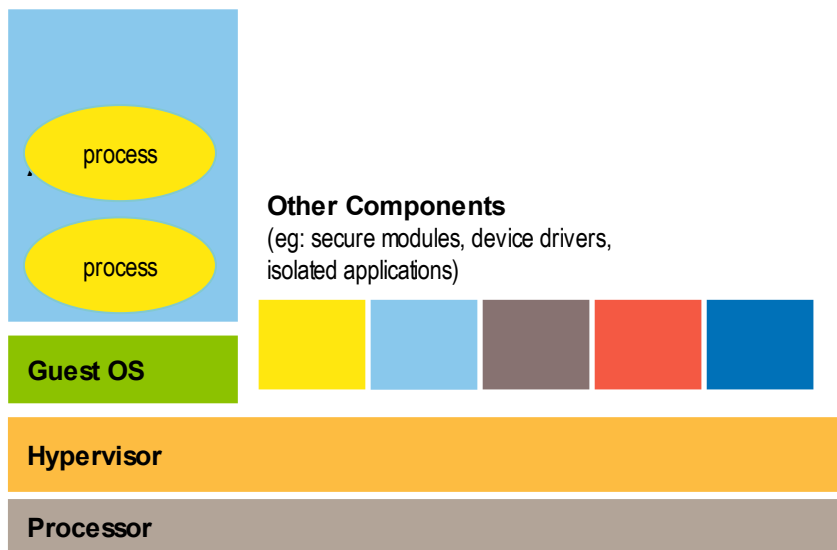


Figure 2.2.

A componentized system. The hypervisor runs a guest operating system as an isolated, course-grained component along with several fine-grained individual components such as secure modules or device drivers.

An ideal componentization solution has the following characteristics:

1. Software system components are isolated from one another in protected execution environments;
2. Components can be constructed with a choice of granularity, from full virtual machines with guest operating systems to much lighter weight execution environments;
3. Decomposition of complex existing software can be done incrementally and to any extent appropriate for a given project.

On its own, virtualization does not enable componentization. Traditional virtualization approaches – as are common in the enterprise space – are entirely course-grained: the lowest level of component is the virtual machine.

Benefits of Virtualization and Componentization

Processor consolidation and multiple concurrent operating systems
Virtualization supports the concurrent existence and operation of multiple operating systems on the same hardware platform. This is driven by the vastly different requirements of the various subsystems that provide separate aspects of the device's functionality.

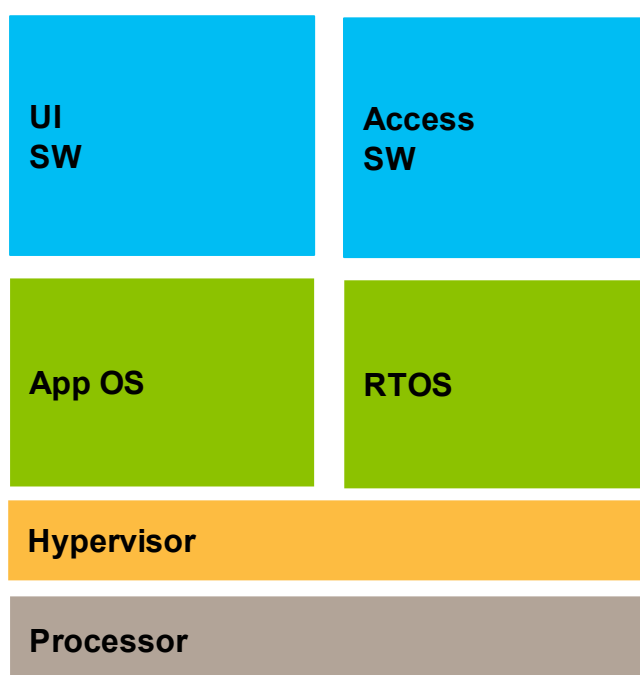


Figure 2.3.

Virtualization allows running multiple operating systems concurrently, serving the different needs of various subsystems, such as real-time environment vs. high-level API.

On the one hand, there is real-time functionality that requires low and predictable interrupt latency. In the case of the mobile phone terminal, the cellular communication subsystem has such real-time requirements. These requirements are traditionally met by a small and highly efficient real-time operating system (RTOS).

On the other hand, there is a large (and growing) amount of high-level application code that is similar (and often identical) to typical application code used on personal computers. Such code is typically developed by application programmers, who are not experts in low-level embedded programming. Such application code is best served by a high-level operating system (also called rich OS, application OS or feature OS) that provides a convenient, high-level application interface. Popular examples are Linux and embedded versions of Windows.

Virtualization serves those different requirements by running appropriate operating systems concurrently on the same processor core, as shown in Figure 3.1. The same effect can be achieved by using separate cores for the real-time and application software stacks. But even in this case, virtualization provides advantages in terms of the security implications of multi-core chips discussed below.

The ability to run several concurrent operating systems on a single processor core can reduce the bill of materials, especially for lower-end devices. It also provides a uniform OS environment in the case of a product series (comprising high-end devices using multiple cores as well as lower-end single-core devices).

Software architecture abstraction

The combination of virtualization and componentization provides both an architectural abstraction of the underlying hardware and a methodology for the development of interchangeable and reusable system components. This provides considerable advantages to manufacturers, system designers, and software developers.

For manufacturers and system designers, the architectural and resource abstraction provides *freedom*. For example, the system designer now has the freedom to map two subsystems onto individual processor cores (with shared physical memory), or share a single core between them, depending on the processing needs of the particular product. The resource abstraction provided by the hypervisor means that most software is unaffected by such a change.

For software developers, the system abstraction provides *future-proofing* and a *market-vehicle* for their software. Developers build applications against a stable API that is protected from changes in the underlying system model, and package such applications into reusable and deployable components that can slot into any system modeled underneath that API.

This combination of advantages can provide significant benefits in both engineering costs, reduced time to market, and reduced BOM costs for a product series consisting of models of different hardware and software capabilities.

Dynamic processor allocation

The same architectural abstraction provided by virtualization and componentization can also be used for a more dynamic partitioning of resources. For example, during periods of high processing demands of the UI software, the high-level OS (assuming it is multiprocessor-capable) can be given a share of the processor normally used for real-time processing, or, during periods of low overall demand, both systems can be migrated to the same core, shutting down the other core completely to save power.

Security

Virtualization and componentization can be used to enhance security. A virtual machine or component encapsulates a subsystem, so that its failure cannot interfere with other subsystems or components. In a mobile phone handset, for example, the communication stack is of critical importance. If the communication stack was subverted by an attacker, the phone could interfere with the network by violating communication protocols. In the extreme case, the phone could be turned into a jammer which disables communication in the whole cell. Similarly, an encryption component needs to be strongly shielded from compromise to prevent leaking of the information the encryption is supposed to protect.

This is a significant challenge for a system running millions of lines of code, which inevitably contain tens of thousands of bugs, many of them security-critical. Especially in an open system, which allows owners to download and run arbitrary programs, the high-level OS is subject to attacks, and is large enough (hundreds of thousands of lines of code) to contain of the order of a thousand bugs. In the absence of virtualization, the high-level OS runs in privileged mode, and therefore, once compromised, can attack any part of the system.

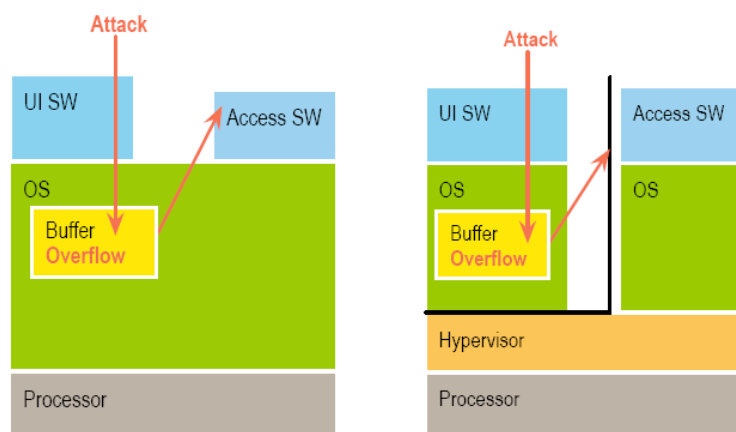


Figure 2.4.

Once the operating system is compromised (e.g. by an application program which exploits a buffer or stack overflow in the kernel), any software running on top can be subverted, as shown on the left. Encapsulating a subsystem into a VM protects other subsystems from a compromised OS.

With virtualization and componentization, the high-level OS is de-privileged and unable to interfere with data belonging to other operating systems or components (Figure 2.4) and its access to the processor can be limited to ensure that real-time components meet their deadlines.

Multi-core chips

The above threat scenario is not eliminated by running the application OS on a separate processor core. Unless the cores also have separated memory (which complicates system design and makes data transfer between cores expensive), a compromised application OS running in privileged mode can still access other subsystems' data, including kernel data structures.

This can be prevented by virtualization and componentization. The hypervisor partitions physical memory between virtual machines and components, and thereby prevents such interference.

Reliability, fault detection and recovery

Virtualization and componentization is an enabler for the development of systems which provide unprecedented reliability, fault detection, and recovery. The decomposition of the system into separate virtual machines and sub-components provides *fault isolation*: a fatal error, system crash, or unstable execution in one component is completely isolated from all other executing components and, since components are de-privileged, from the hypervisor itself.

This inherent stability also provides for the opportunity to perform fault detection and recovery. Components can be securely and efficiently shut-down and restarted on detection of a component crash, resulting in minimization of the impact on consumers of that component and a seamless user experience.

One particular area of system design, which has traditionally been a cause of system instability, is device driver execution. Research reveals that bug density is highest in device driver code. The secure decomposition of traditional kernel drivers into de-privileged components that virtualization and componentization completely removes the disastrous system-wide impact of a device driver crash. This allows the device to be cleanly restarted with little to no impact on the remainder of the executing system

System management and update

Just as the user-level software on mobile devices is no longer static, nor is the system software. With standards such as the Open Mobile Alliance Device Management (OMA DM), and emerging technologies such as FOTA (Firmware Over-The-Air), methodologies are being developed that allow manufacturers and network operators to remotely manage the system level components on a device. This poses extreme challenges for traditional system software.

Virtualization and componentization can be used to overcome these challenges. Run time management and replacement is enabled via all system level components being virtualized into individual secure components. The hypervisor can control remote authentication to allow new components and management tasks to be delivered and deployed over the air.

Certification re-use

For devices (such as mobile phones) which require certification, the isolation afforded by virtualization and componentization can be used to reduce certification costs. Typically, product innovation is predominantly in the user-facing functionality, which therefore changes much faster than the real-time subsystem (which tends to be the part requiring certification). By isolating the certified subsystem from any changes in the rest of the system, the certification remains valid.

This enables further design choices, including completely opening up the UI part of the devices, allowing users to completely change their OS environment (as done on the OpenMoko phone or a number of MIDs) without the hardware cost of complete physical separation. Some trusted versions of the UI stack could be given higher privileges, as long as the hypervisor is able to validate a certificate of trust.

IP protection

Virtualization and componentization enables the secure isolation and protection of deployed intellectual property and media content. Devices (a concrete, real-life example that will ship this year is a HD IPTV set-top box) may contain code that must be protected against theft (for example, the implementation of proprietary decompression algorithms). The device can easily be dismantled and hardware probes attached, so it must be protected against such attacks.

The solution is to ship the code encrypted, with the decryption key kept in on-chip non-volatile memory. A secure boot process initializes the device to run a certified copy of the hypervisor in on-chip memory, and makes the decryption key available to the hypervisor. The hypervisor then decrypts and loads the proprietary code into a virtual machine which executes from on-chip memory, protected by the hypervisor from any other code running on the system, including the main OS (such as Linux) which operates the device. The main OS communicates with the protected decompression service via communication channels provided by the hypervisor. Similar setups can be used for protecting media contents (digital rights management).

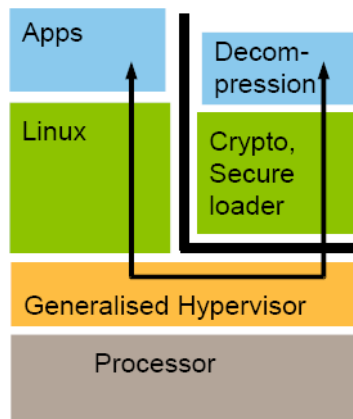


Figure 2.5

The cryptographic algorithm IP is securely protected from the remaining system.

License separation

Linux is a frequently deployed high-level OS. Its advantages are its royalty-free status, independence from specific vendors, widespread deployment and a strong and vibrant developer community and large ecosystem.

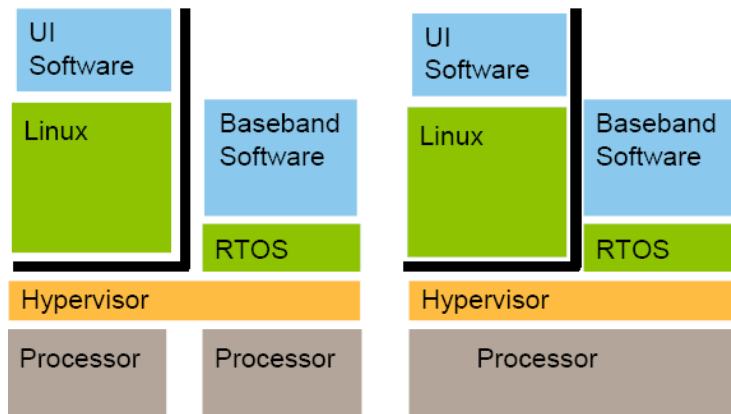
Linux is distributed under the GPL license, which requires that all derived code is subject to the same license, and thus becomes open source. There are legal arguments that this even applies to device drivers that are loaded as binaries at run time into the kernel.

Virtualization is frequently employed to provide a proprietary execution environment for software that is to share the processor with a Linux environment. Linux and the proprietary environment are run in separate virtual machines. A stub (or proxy) driver is used to forward Linux driver requests to the real device driver, using hypercalls.

Example Use-Cases for Virtualization and Componentization

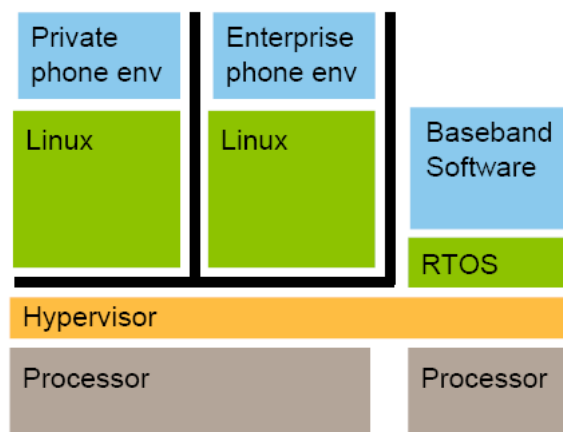
Open phone with user-configured OS

Virtualization and componentization enables the development of a phone distributed with a user-configurable high-level operating system (such as Linux), providing an open development environment that can leverage a rich application ecosystem. Componentization enables the secure separation of the legacy RTOS and base-band stack from the open and insecure environment of the user-configured OS.



Phone with private and enterprise environment

Virtualization and Componentization enables the development of a phone with two securely partitioned environments: an enterprise application environment tightly controlled for business use (which integrates with the enterprise IT system), and an open environment for private use (which holds private data, such as address books and bank account access codes and allows the user to install software of their choice). Each environment runs its own operating system (they could be different, e.g. Windows and Linux) and the environments can be switched by pressing a key. Due to the mutual distrust between the two environments, strong isolation is a requirement of the underlying virtualization solution.



Issues with traditional approaches in embedded systems

It is clear that virtualization and componentization can address the stringent requirements placed by the increasing mobile and CE market pressures on embedded systems development.

There are, of course, existing solutions in the marketplace that attempt to meet these requirements. What we find, however, is that these solutions are only ever capable of piecing together one part of the puzzle: they either satisfy virtualization, via coarse-grained virtual machines, or componentization, via high-level user libraries, but never both. These solutions are generally fine in enterprise markets in which resource constraints are not an issue. In embedded systems development, however, we assert that it is only via an effective combination of virtualization and componentization that the challenges unique to this market will be overcome.

Virtualization is of very little help in reducing the complexity of the device. The isolation provided by virtualization is by its nature coarse-grain — it provides the illusion of a complete machine for each subsystem. This means that each virtual machine is required to run its own operating system, making them relatively heavyweight. Increasing the number of virtual machines in order to reduce the granularity of the subsystems would create serious performance issues, and significantly increase the amount of code. This, in turn, not only requires increased memory size (and thus power consumption) but also more points of failure.

Unlike a server that uses virtualization to run many independent services in their own virtual machines, embedded systems are highly integrated. Their subsystems are all required to co-operate closely in order to achieve the overall device functionality. This tight co-operation requires highly-efficient communication between subsystems, characterized by high bandwidth and low latency. This is the antithesis of the virtual-machine model, where each VM is considered a system of its own, which communicates with other systems via file systems or networks. The kind of communication required between components of an embedded system requires shared memory and low-latency signaling, requirements that simply do not fit the virtual-machine model.

The integration requires sharing of physical devices, which must be accessed (in a strictly controlled fashion according to some sharing policy) by different subsystems. A virtualization-only approach supports running device drivers in their native (guest) OS, but that means that a device is owned by a particular guest, and not accessible by others, and that the guest is trusted to drive the particular device.

Furthermore, many embedded systems must meet critical security requirements. Virtualization alone does not help in addressing these requirements. While it is essential that subsystems can communicate effectively and efficiently where needed, communication must be disabled where it is not needed or could lead to leakage of critical information. For example, bank-account access keys must be protected from disclosure, and licensed media content must be protected from copying. A security-capable componentization solution is required.

Critically, such a componentization solution must be deployed on as small a TCB as possible. Virtualization does not support minimizing the TCB. Compared to running a service on top of a native OS, running it in a virtual machine requires a hypervisor and a guest OS, both part of the TCB. Compared to a native OS, virtualization increases the TCB. What is required is a framework in which trusted services can be built with a dependency on a minimal amount of other code. At the same time, the trusted service frequently has high performance requirements too, meaning that it must be able to communicate efficiently with the rest of the system. Virtualization does not serve this requirement.

THE OKL4 APPROACH

VIRTUALIZE AND COMPONENTIZE WITH SECURE HYPERCELL TECHNOLOGY

OKL4 and the Secure HyperCell™ Technology

OKL4 is Open Kernel Labs (OK Labs) operating system, embedded hypervisor, virtualization and componentization platform. At its core is the OKL4 microkernel, the commercially-distributed and commercially-supported member of the L4 microkernel family.

OKL4 is the world's most advanced commercial microkernel system, based on 14 years of research leadership in the microkernel area, and hardened by several years of commercial deployments.

Central to the success of OKL4 in overcoming the challenges in the embedded space is OK Labs unique *Secure HyperCell Technology*. The Secure HyperCell is the architectural container for your next-generation embedded system: a development, deployment, and execution engine for both course-grained virtual machines and fine-grained embedded components, each securely isolated in individual OKL4 Cells.

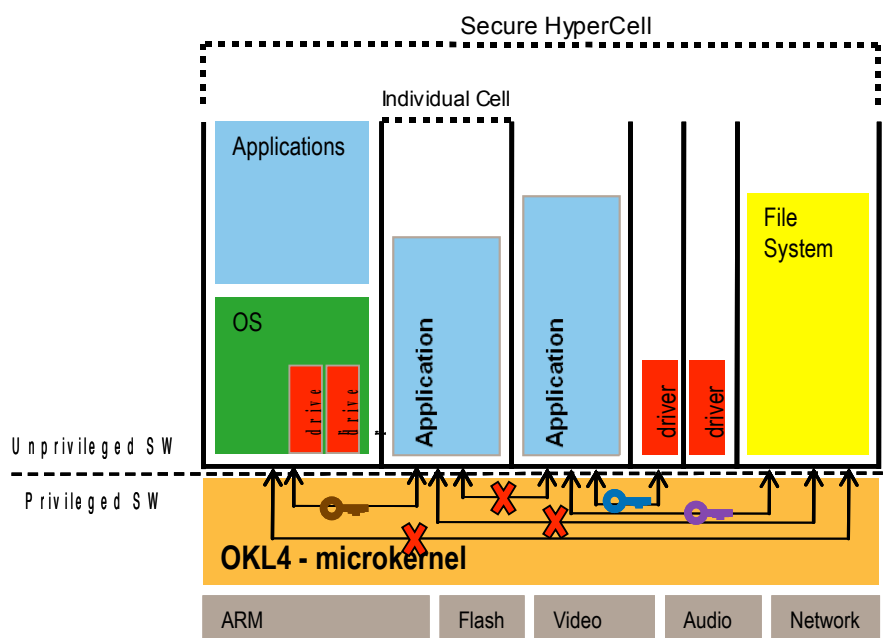


Figure 3.1
Structure of an OKL4 Secure HyperCell.

Fundamental features of OKL4 Secure HyperCell Technology

Microkernel based

In order to best address the challenges discussed above, we would need a technology that has the following properties:

1. support for virtualization with all its benefits;
2. support for lightweight but strong encapsulation of medium-grain components that interact strongly, in order to build robust systems that can recover from faults;
3. high-bandwidth, low-latency communication, subject to a configurable, system-wide security policy;
4. ability to build subsystems with a very small trusted computing base.

Microkernel technology provides the ideal foundation for meeting the above requirements. A microkernel is defined by Liedtke's minimalism principle:

A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.

This minimality implies that a microkernel does not offer any services, only the mechanisms for implementing services. Actual system services are implemented as components running in (unprivileged) user mode. As such, a microkernel implements the principle of separation of policy and mechanism: the kernel provides mechanisms that allow controlling resources, but the policies according to which resources are used are implemented in user-mode system components.

The microkernel approach leads to a system structure that differs significantly from that of classical "monolithic" operating systems, as shown in Figure 3.2. While the latter have a vertical structure of layers, each abstracting the layers below, a microkernel-based system exhibits a horizontal structure. System components run beside application code, and are invoked by sending messages.

A main characteristic of a well-designed microkernel is that it provides a generic substrate on which arbitrary systems can be built, from virtual machines to highly-structured systems consisting of many separate (but interacting) components.

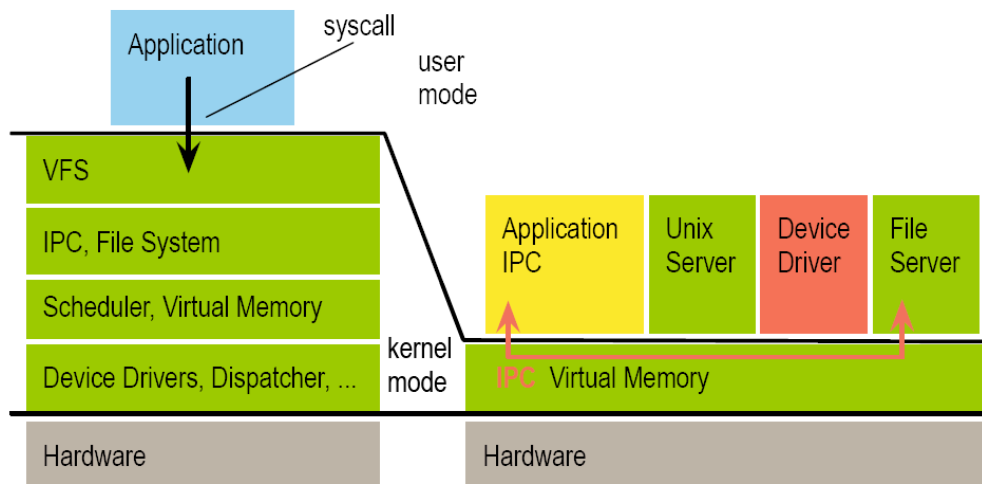


Figure 3.2
Structure of monolithic and microkernel-based systems.

Low overhead virtualization

For more than ten years L4 has been successfully used as a hypervisor for virtualizing Linux. While the approach used is essentially that employed years earlier by Mach, L4's vastly better IPC performance allowed it to succeed where Mach-based virtualization failed owing to intolerable overheads. The performance of OKL4-based virtual machines depends somewhat on the underlying processor architecture, but is generally within a few percent of the native performance. This overhead is about the same as that achieved by specialized hypervisors that lack the generality of the OKL4 platform.

A particularly interesting result is that of Linux virtualized on ARMv5 platforms. Here OK Linux (Linux para-virtualized on OKL4) outperforms native Linux in Imbench context-switching and other microbenchmarks, by factors of up to 50. This seemingly paradoxical result bears witness to the expertise of the OK Labs kernel team. However, it also reflects the fact that it is much easier to thoroughly optimize a small code base of around 10,000 lines than a system of the size of the Linux kernel.

Figure 3.3 shows the structure of OK Linux. The hardware-abstraction layer (HAL) of Linux is replaced by a version that maps to the OKL4 "architecture". This OKL4-HAL is in fact mostly independent of the underlying processor architecture.

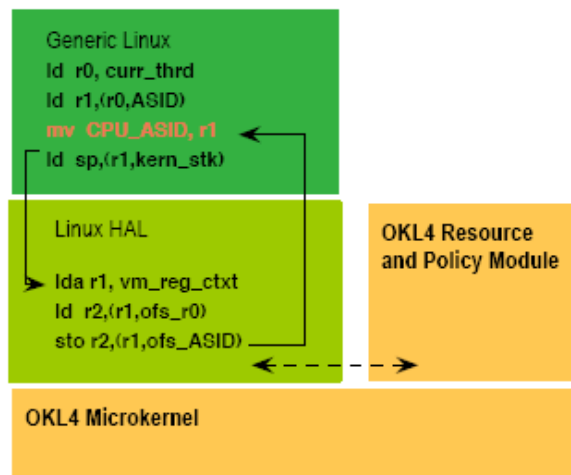


Figure 3.3
Virtualization in OK Linux

A virtualization event is primarily handled inside the HAL: Either a sensitive instruction traps, invoking the hypervisor (OKL4 microkernel), which reflects the trap back into the HAL. Or the sensitive instruction was para-virtualized into a direct jump to virtualization code in the HAL. The virtualization code then returns directly to the instruction following the virtualized one.

The HAL can handle some virtualizations directly because it holds copies of some virtual machine state, the real state is held outside the Linux kernel's address space for security reasons. Even where a virtualized instruction changes VM state, it is often possible to perform this action on the local copy, and synchronizing with the master copy lazily on certain events (e.g. when the VM's time slice expires).

Some virtualization events require a synchronous change of the virtual state, e.g. where this changes the physical resource allocations. In such a case, the HAL invokes the resource and policy-management module via an IPC message.

Unbeaten IPC performance

The key to performance of any system built on OKL4 is the high performance of its message-passing IPC mechanism. This is also the enabler for low-overhead virtualization: A system-call trap executed by a guest application in a virtual machine invokes the microkernel's exception handler, which converts this event into an IPC message to the guest operating system. The guest handles it like a normal system call. The system-call result is returned back to the guest application via another IPC message, which unblocks the waiting guest process.

Similarly, IPC is used to deliver interrupts to the guest OS's interrupt handler. It is also used to communicate with device drivers, and for communication and synchronization between any components of the system, including between virtual-machine environments.

As the same mechanism is used for many different operations, it is highly optimized. Optimizing IPC implicitly optimizes the mechanism behind most critical system operations. As it is a relatively simple mechanism, it is possible to optimize it completely in virtually all of its aspects.

IPC performance has been the hallmark of OKL4 and its predecessor L4 kernels since the beginning. *IPC performance data for those kernels has been published for years, and has never been beaten by other kernels.*

Capability-based security model

OKL4 supports mandatory access control based on an underlying fine-grained model using capabilities. OKL4 capabilities are kernel objects which control all other kernel objects, namely virtual and physical memory (including device registers), address spaces, threads, and IPC. Capabilities can thus be used to delegate authority over resources to subsystems — a Cell is a subsystem associated with a particular set of capabilities. CPU time is controlled via scheduler threads using capability mediated mechanisms to allocate time slices to threads.

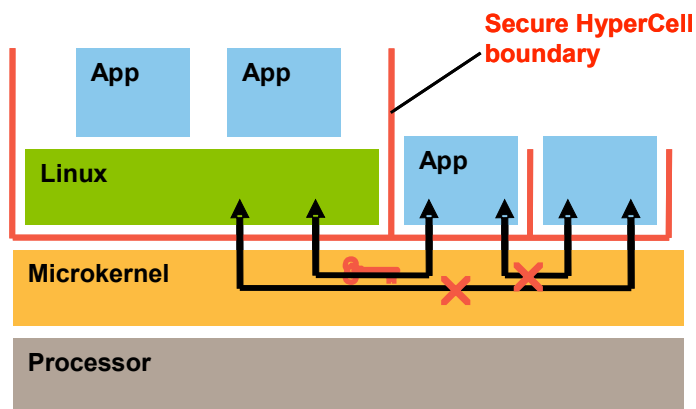


Figure 3.4
Capabilities used for information-flow control between Secure HyperCells

Capabilities are therefore a mechanism to enforce mandatory information-flow control between Cells. As indicated in Figure 3.4, information flow is allowed between cells if they have a capability for an information channel (IPC endpoint or shared memory), and prevented otherwise. This allows implementing virtually arbitrary security policies.

Efficient resource sharing

OKL4 provides mechanisms for efficient sharing of resources. Arbitrary memory regions can be shared by setting up mappings between address spaces. This is generally used to provide high-bandwidth communication channels between subsystems. Shared memory regions can be created with appropriate permissions. For example a buffer shared between processes in a producer-consumer relationship can be made accessible to the consumer read-only.

A typical scenario of communication via shared buffers is I/O via high-bandwidth devices. The device driver shares a buffer with a client in order to provide zero-copy I/O operations. Another case of resource sharing is joint access to devices from separate subsystems, including virtual-machine environments. For example, a Linux system running in a virtual machine may need to access a device (touch screen, audio) that is also required by other subsystems.

As shown in Figure 3.5, a shared device will have a device driver which may live inside a virtual-machine environment, or in its own address space. The former allows reuse of the guest OS's native drivers (e.g. an unmodified Linux driver can be used), while the latter provides better security, as the driver is isolated from other code, leading to better fault isolation. In any case, *device drivers in OKL4 always run in user mode* (unless the hardware platform requires privileged execution).

In such a scenario, other subsystems can access the device by communicating with the driver via an IPC protocol. In a virtual machine, this is achieved by inserting a *proxy driver* into the guest OS, which converts I/O commands into IPC messages to the real driver.

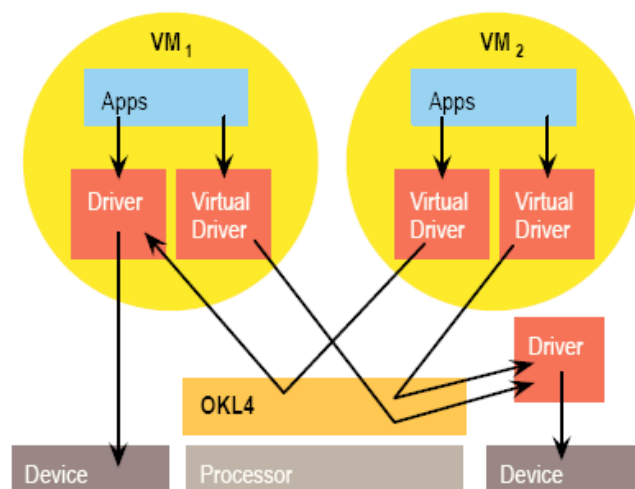


Figure 3.5

Devices can be efficiently shared between virtual-machine environments, by the use of stub drivers. The real driver can either reside in a virtual machine (e.g. a native Linux driver) or run directly on OKL4 in its own protected address space.

Componentization and lightweight execution environments

Conventional virtualization solutions only support running complete operating systems in a virtual machine. While OKL4 supports low-overhead virtualization in this course-grained manner, the Secure HyperCell architecture also allows development and deployment of arbitrary system components in lightweight execution environments (LWEE).

An LWEE executes an application or system service directly on OKL4 without inclusion of a complete operating system. Critical services software can be hosted directly on top of the OKL4 microkernel with no dependence on a guest OS.

This fine-grained isolation of components is an enabling feature of many of the benefits discussed above. Security-sensitive and proprietary software can be separated from an Application OS (e.g. Linux) without running a second instance of an Application OS. DSP software components (e.g. multimedia codecs) can be migrated to host processors without sacrificing real-time guarantees or increasing power consumption. Remote system management and update of components via systems such as FOTA are significantly easier to achieve. Fine-grained access control between components is enabled by the generalized capability security mechanism.

Small trusted computing base

By keeping as much code as possible out of the kernel, the kernel itself can be made very small, around 10,000 lines, without restricting its universality. In fact, the strict separation of mechanisms (in the kernel) and policies (in user-mode components) ensures that the kernel can be used in arbitrary application scenarios and industry verticals.

A really big advantage of the small size of the kernel is that it allows minimization of the amount of code that must be trusted, i.e., the system's trusted computing base. In contrast to plain virtualization approaches, which are designed to be always used with a guest OS underneath any other software, the amount of trusted user-mode code can be kept much smaller.

With OKL4, a minimal trusted computing base consists of the kernel, the user-mode policy module, and possibly some library code as required to support the security-critical code. The total TCB of a critical application can be kept as small as 15,000 lines, while concurrently running a large amount of untrusted code.

Summary

The mobile and CE marketplace is experiencing a rapid evolution towards user-customizable, open software environments with a greater demand on security, reliability, performance, time-to-market, and BOM costs than ever before. These demands create severe challenges in the engineering of embedded systems. We must change the way we engineer, and we must engineer the change.

Virtualization and componentization provide unique capabilities to system designers, device manufacturers, and software developers to overcome these challenges. Existing solutions in the marketplace, however, employ virtualization techniques transplanted from the enterprise server market which fail under the unique constraints of embedded systems. An effective solution must combine the advantages of both virtualization and componentization in a system specifically engineered for these constraints.

We presented OKL4 with Secure HyperCell technology, a microkernel-based embedded hypervisor solution providing low-overhead virtualization and secure componentization. Combining high performance, capability-based security, efficient resource sharing, unprecedented reliability, and lightweight execution environments delivered over a greatly minimized Trusted Computing Base, OKL4 provides the platform for engineering more fault resilient, secure, and easily maintained embedded systems for the next generation of mobile and CE devices.

About Open Kernel Labs

The Open Kernel Labs leading technology in embedded systems software and virtualization enables the development of safe, trustworthy and affordable devices. Backed by the largest, independent team of microkernel developers, OK Labs delivers OKL4, an advanced microkernel solution, which offers the highest performance combined with strong protection and security features. OKL4 provides developers with a robust, open source platform for building secure, differentiated embedded applications. For more information about OK Labs and its products visit www.ok-labs.com. OK Labs is a spin out from NICTA, Australia's preeminent Center of Excellence for information and communications technology, <http://nicta.com.au/>.

For information on the OK Community, please visit the Community Portal at www.ok-labs.com/community/community-portal. Participants can join the Developer's Mailing List at <http://www.ok-labs.com/community/mailling-list-signup>.

Open Kernel Labs, OK Labs and Secure HyperCell are trademarks or registered trademarks of Open Kernel Labs or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners. Open Kernel Labs, OK Labs and Secure HyperCell are trademarks or registered trademarks of Open Kernel Labs or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.