Embedded Systems Security - Part 4: I/O virtualization

David Kleidermacher and Mike Kleidermacher - February 25, 2013

Editor's Note: Embedded Systems Security aims for a comprehensive, systems view of security: hardware, platform software (such as operating systems and hypervisors), software development process, data protection protocols (both networking and storage), and cryptography. In this excerpt, the authors offer an in-depth look at the role of the operating system in secure embedded systems. In part 1, the authors offer an in-depth look at the role of the operating system in secure embedded systems. In part 2, the authors discuss how an OS provides access control for ensuring process security. In part 3, the authors examine the use of hypervisors in implementing system virtualization. In this installment, the authors review the security pitfalls and trends in embedded I/O virtualization.

Adapted from "Embedded Systems Security" by David Kleidermacher and Mike Kleidermacher (Newnes)

2.7 I/O Virtualization

Key Point
One of the biggest impacts on security and efficiency in system virtualization is the approach to managing I/O across virtual machines.

This section discusses some of the security pitfalls and emerging trends in embedded I/O virtualization.

2.7.1 The Need for Shared I/O
In any embedded system, invariably there is a need for sharing a limited set of physical I/O peripherals across workloads. The embedded operating system provides abstractions, such as layer two, three, and four sockets, for this purpose. Sockets provide, in essence, a virtual interface for each process requiring use of a shared network interface device. Similarly, in a virtualized system, the hypervisor must take on the role of providing a secure virtual interface for accessing a shared physical I/O device. Arguably the most difficult challenge in embedded virtualization is the task of allocating, protecting, sharing, and ensuring the efficiency of I/O across the virtual machines and applications.

2.7.2 Emulation
The traditional method of I/O virtualization is emulation: all guest operating system accesses to device I/O resources are intercepted, validated, and translated into hypervisor-initiated operations (see Figure 2.25). This method maximizes reliability, security, and shareability. The guest operating system can never corrupt the system through the I/O device because all I/O
accesses are protected via the trusted hypervisor device driver. A single device can easily be multiplexed across multiple virtual machines, and if one virtual machine fails, the other virtual machines can continue to utilize the same physical I/O device, maximizing system availability. The biggest drawback is efficiency; the emulation layer causes significant overhead on all I/O operations. In addition, the hypervisor vendor must develop and maintain the device driver independent of the guest operating system drivers.

### 2.7.3 Pass-through

In contrast, a pass-through model (see Figure 2.26) gives a guest operating system direct access to a physical I/O device. Depending on the CPU, the guest driver can either be used without modification or with minimal paravirtualization. A single device can be shared between multiple guests by providing a virtual I/O interface between the guest that owns the physical device and any other guests that require access to that device. For network devices, this virtual interface is often called a virtual switch (layer 2) and is a common feature of most hypervisors. The pass-through model provides improved efficiency but trades off robustness: an improper access by the guest can take down any other guest, or application, or the entire system. This model violates the primary security policy of system virtualization: isolation of virtual environments for safe coexistence of multiple operating system instances on a single computer.
If present, an IOMMU enables a pass-through I/O virtualization model without risking direct memory accesses beyond the virtual machine’s allocated memory. As the MMU enables the hypervisor to constrain memory accesses of virtual machines, the IOMMU constrains I/O memory accesses (especially DMA), whether they originate from software running in virtual machines or the external peripherals themselves (see Figure 2.27).

Title-1

IOMMUs are becoming increasingly common in embedded microprocessors, such as Intel Core, Freescale QorIQ, and ARM Cortex A15. Within Intel processors, the IOMMU is referred to as Intel Virtualization Technology for Directed I/O (Intel VT-d). On Freescale’s virtualization-enabled QorIQ processors such as the P4080, the IOMMU is referred to as the Peripheral Access Management Unit (PAMU).

On Cortex A15 (and other ARM cores that support the ARM Virtualization Extensions), the
IOMMU is not part of the base virtualization specification. Rather, ARM Ltd. has a separate intellectual property offering, called a System MMU (SMMU), which is optionally licensable by ARM semiconductor manufacturers. In addition, the manufacturer may use a custom IOMMU implementation instead of the ARM System MMU. In addition, ARM TrustZone provides a form of IOMMU between the normal and secure zones of an ARM processor; normal zone accesses made by the CPU or by peripherals allocated to the normal zone are protected against accessing memory in the secure zone.

The IOMMU model enables excellent performance efficiency with increased robustness relative to a pass-through model without IOMMU. However, IOMMUs are a relatively new concept.

<table>
<thead>
<tr>
<th>Key Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>A number of vulnerabilities (ways to circumvent protections) have been discovered in IOMMUs and must be worked around carefully with the assistance of a systems software/hypervisor supplier.</td>
</tr>
</tbody>
</table>

In most vulnerability instances, a faulty or malicious guest is able to harm security via device, bus, or chipset-level operations other than direct memory access. Researchers at Green Hills Software, for example, have discovered ways for a guest operating system to access memory beyond its virtual machine, deny execution service to other virtual machines, install a malicious hypervisor below the default system hypervisor (an attack described later in this chapter), and take down the entire computer via IOMMU-protected I/O devices.

For high-reliability and/or security-critical applications, the IOMMU must be applied in a different way than the traditional pass-through approach in which the guest operating system has unfettered access to the I/O device. The myriad of trade-offs for use of the IOMMU are beyond the scope of this book; consult a hypervisor vendor to understand the options for use of the IOMMU and I/O virtualization in general.

A major downside of a pass-through approach (with or without the IOMMU) is that it prevents robust sharing of a single I/O device across multiple virtual machines. The virtual machine that is assigned ownership of the pass-through device has exclusive access, and any other virtual machines must depend on the owning virtual machine to forward I/O. If the owning virtual machine is compromised, all virtual machines will be denied servicing for that device.

2.7.4 Shared IOMMU

This deficiency has led to emerging technologies that provide an ability to share a single I/O device across multiple guest operating systems using the IOMMU and hardware partitioning mechanisms built into the device I/O complex (e.g., chipset plus the peripheral itself). One example of shareable, IOMMU-enabled, pass-through devices is Intel processors equipped with Intel Virtualization Technology for Connectivity (Intel VT-c) coupled with PCI-express Ethernet cards implementing Single-Root I/O Virtualization (SR-IOV), a PCI-SIG standard.

With such a system, the hardware takes care of providing independent I/O resources, such as multiple packet buffer rings, and some form of quality of execution service among the virtual machines. This mechanism lends itself well to networking devices such as Ethernet, Rapid I/O, and Fibre Channel; however, other approaches are required for secure, independent sharing of peripherals such as graphics cards, keyboards, and serial ports. Nevertheless, it is likely that hardware-enabled, IOMMU-protected, shareable network device technology will grow in popularity across embedded processors.
2.7.5 IOMMUs and Virtual Device Drivers
Earlier in this chapter, we discussed the importance of virtual device drivers in limiting privilege and promoting robust system design. The ideal virtual device driver requires very little device-specific code to reside within the supervisor-mode kernel: the interrupt service routine (ISR) and, in the case of network devices, access to direct memory access (DMA) programming registers. The ISR must be in the kernel (the interrupt vector is executed by the hardware in supervisor mode). The DMA programming is often kept in the kernel because the operation must be trusted: access to DMA registers enables the driver to overwrite any physical memory location, even the kernel itself.

Unfortunately, the virtual driver approach still leaves a bit of device-specific code in the kernel. For improved maintainability and a cleaner architecture, it would be better if the DMA programming could reside in user mode without increasing the driver’s privilege. An IOMMU enables the DMA programming to reside in the virtual device driver. Perhaps most importantly, by enabling direct access to memory-mapped device registers and DMA programming, the IOMMU promotes a purer form of virtual device driver architecture without sacrificing performance efficiency.

2.7.6 Secure I/O Virtualization within Microkernels
As discussed earlier, virtual device drivers are commonly employed by microkernel-style operating systems. Microkernel-based hypervisors are also well suited to secure I/O virtualization: instead of the typical monolithic approach of placing device drivers into the hypervisor itself or into a special-purpose Linux guest operating system (the Dom0 method described earlier), the microkernel-based hypervisor uses small, reduced-privilege, native processes for device drivers, I/O multiplexors, health managers, power managers, and other supervisory functions required in a virtualized environment. Each of these applications is provided only the minimum resources required to achieve its intended function, fostering secure embedded system designs. Figure 2.28 shows the system-level architecture of a microkernel-based hypervisor used in a multicore networking application that must securely manage Linux control plane functionality alongside high-throughput, low-latency data plane packet processing within virtual device drivers.

Without virtualization, the preceding application could be implemented with a dual Linux/RTOS configuration in which the control and data plane operating systems are statically bound to a set of independent cores. This is referred to as an Asymmetric Multiprocessing (AMP) approach. One advantage of virtualization over an AMP division of labor is the flexibility of changing the allocation of control and data plane workloads to cores. For example, in a normal mode of operation, the architect may want to use only a single core for control and all other cores for data processing. However, the system can be placed into management mode in which Linux needs four cores (SMP) while the data processing is temporarily limited. The virtualization layer can handle the reallocation of cores seamlessly under the hood, something that a static AMP system cannot support.
Security can also be improved by adding applications, or even separate virtual machines (the virtual appliances concept described earlier in this chapter), which perform a dedicated security function such as anti-malware or firewalling.

**Title-3**

Increases in software and system complexity and connectivity are driving the evolution in how embedded systems manage I/O and in the architecture of the operating systems and hypervisors that are responsible for ensuring their security. The combination of a reduced-privilege, component-based designs as well as intelligent I/O virtualization to enable secure consolidation without sacrificing efficiency will remain a focus of systems software suppliers in meeting the flexibility, scalability, and robustness demands of next-generation embedded systems.

### 2.8 Remote Management

When an embedded system fails in the field, developers (and sometimes government forensics teams) are tasked with determining the cause of failure. A flight recorder is a well-known field diagnostic system: the end product (airplane) is shipped with a built-in diagnostic capability (the black box). Yet a burgeoning class of embedded devices requires field diagnostic and management capabilities. Unlike the black box, which is a purely forensic tool, an active network connection is required in many systems. This connection enables technicians to inspect a fielded system to locate the source of anomalous behavior such as loss of function or performance degradation, install patches or other software upgrades, perform automated audits, change configuration, or execute a plethora of other management duties.

Furthermore, with the increasing availability of network services built into embedded systems, device management can be conveniently discharged via the Internet. A home’s cable or satellite box has a network connection that most likely has been used to carry out both remote diagnostics and firmware upgrades. Device management functionality has been transformational, increasing product lifetime, reliability, serviceability, and customer satisfaction while reducing maintenance cost and total cost of ownership.

A great example of the power of remote management is the Mars Pathfinder: remote management saved the 1997 mission from disaster when a malfunction was diagnosed down to a software defect that was remedied with a patch installed via radio link from Earth.

#### 2.8.1 Security Implications

The hacker’s ambition: locate a vulnerability that, when properly manipulated, provides access into a computer system for nefarious purposes. Over time, remote network exploits have become increasingly sophisticated. In April 2010, IBM security researcher Mark Dowd won acclaim with his 25-page report detailing an astoundingly convoluted set of steps that could be taken to exploit a web access vulnerability, previously believed to be innocuous, in Adobe’s ubiquitous Flash program.15

<table>
<thead>
<tr>
<th><strong>Key Point</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote device management is the answer to the hacker’s wildest dreams: the embedded system is imbued not only with Internet access, but also a means to remotely modify and patch software.</td>
</tr>
</tbody>
</table>

No Byzantine attack vector required; just get past the basic operating system controls, and the embedded device becomes a playground of iniquity.

In Chapter 1, we briefly described the VxWorks remote management vulnerability in which an
operating system diagnostics port is commonly left open for access by any novice hacker to exploit. This vulnerability is so widespread that it is unlikely the Internet will ever be fully scrubbed of devices containing this flaw. In the case of the VxWorks flaw, the remote diagnostics connection trivially enables a hacker to install malware, even rooting or replacing the operating system itself (see Figure 2.29).

![Figure 2.29. Malware insertion into embedded system via remote management port.](image)

A basic defense is to guard the remote management port with strong authentication, using standard network security protocols such as TLS/SSL or IKE/IPsec (both described in Chapter 5). First, the remote embedded system must authenticate the computer used by the remote administrator. This ensures that only known, trusted administrative computers are used to access the embedded system for management purposes. Second, if management commands are invoked by a human operator, then the operator must be strongly authenticated to the management computer locally prior to establishing the remote connection. Once the operator is authenticated, SSL or IPsec will use authenticated encryption to protect the integrity and confidentiality of remote management commands and data.

Of course, hacking the management computer may render useless the aforementioned network security protections. For example, malware within the host computer’s operating system can piggyback over the encrypted connection to infiltrate the embedded system. Extreme care must be taken to protect the management computer from the Internet or other open network accessible to hackers. Ideally, management workstations are dedicated to their purpose and never connected to the Internet.

Using an insecure operator workstation to administer a remote embedded system is akin to putting a padlock on a safe built from cardboard. One example is found in arguably the most famous remote management system in the world: Windows Update. Windows Update is designed to unobtrusively and remotely feed a personal computer with the latest validated security patches. Yet hackers have commandeered this facility to upload unauthorized software.

An SSL connection is only the tip of the iceberg in ensuring that embedded systems are developed properly to enable secure remote management. Insider threats and development flaws can cause an embedded system to be fielded with vulnerabilities that turn a remote management channel into a powerful hacker access method. A secure development process to address these threats is a focus of Chapter 3. Without proper assurance that includes testing at the binary level, secure delivery, and other controls, developers can insert back doors using a wide range of proven techniques. A secure device management solution prevents malicious code insertion by employing high-assurance authentication and digital signatures: unauthorized IT administrators, technicians, janitors, and users cannot circumvent the mandatory access controls imposed by the system.

While developers can employ a high-assurance development process for their own software, how can
developers protect against vulnerabilities in third-party operating systems, many of which are shipped in binary form only and lack any security provenance, pedigree, or indemnification? Since the operating system often provides the network security capabilities described earlier, this is obviously a critical issue.

**Key Point**

System virtualization can provide an effective solution to the problem of incorporating or retro-fitting secure remote management to embedded systems: the legacy operating environment is uplifted into a virtual machine, securely isolated from remote management functions, such as connection authentication and configuration management, provided by a trustworthy hypervisor.

In fact, the device management software can be used to monitor, configure, and patch the legacy operating system kernel itself (see Figure 2.30).

![Device management architecture for legacy systems.](image)

In many cases, a secure device management solution involves consulting services to ensure that the appropriate set of security components is integrated and deployed in a robust and cost-effective manner into end devices. Given the increasing financial, safety, and security risks associated with remote access, many embedded and mobile device makers are rethinking their device management strategy. System virtualization is one potentially powerful approach to addressing the secure remote management challenge.

### 2.9 Assuring Integrity of the TCB
#### 2.9.1 Trusted Hardware and Supply Chain

We now turn to a critical systems-level security threat that can render impotent even the most perfectly implemented, absolutely secure software. Embedded system security requires integrity of the TCB (trusted operating system and any trusted applications and middleware, including data protection protocols discussed in Chapter 5).

**Title-5**

For example, a virtual memory operating system depends on the fact that no unauthorized entities can access memory that the operating system allocates to a specific process using the
microprocessor memory management and protection hardware. If the operating system is booted up on malicious hardware that exposes certain memory locations (e.g., by sending the contents of that memory over an attached Ethernet interface), then obviously the system security policies enforced by the operating system are moot.

Attacks on hardware components, subsystems, and peripherals are not far fetched. Numerous reports of counterfeit and subverted hardware have occurred over the years. In 2007, some Maxtor/Seagate hard drives were found to have a pre-installed (by the contract manufacturer) virus that would send data stored on the drive to malicious websites. A hidden “kill switch” inserted into a field-programmable gate array (FPGA) during its design was allegedly used to disable radar systems during a military attack.

Key Point
Embedded system developers must do all they can to ensure the trustworthiness of the hardware supply chain.

Supply chain and manufacturing security is a complicated topic in itself and is beyond the scope of this book. The reality is that most embedded systems organizations do the best they can by purchasing from reliable suppliers and take some risk with respect to trusting hardware.

2.9.2 Secure Boot

Key Point
Beyond trusted hardware, we must ensure that the trusted firmware/software cannot be subverted during the boot process.

Platform firmware attacks are far less difficult and expensive to perpetrate than supply chain attacks and represent an important threat that all embedded systems developers must consider. The act of establishing a secure initial state is often referred to as secure boot.

If the hardware and boot loader have the capability to load the system firmware (operating system, hypervisor, entire TCB) from an alternative device, such as USB, rather than the intended, trusted device (e.g., Flash), then an attacker with access to the system can boot an evil operating system that may act like the trusted operating system but with malicious behavior, such as disabling network authentication services or adding backdoor logins.

Alternatively, an evil hypervisor can be booted, and the hypervisor can then launch the trusted operating system within a virtual machine. The evil hypervisor has complete access to RAM and hence can silently observe the trusted environment, stealing encryption keys or modifying the system security policy. King et al., provide a good example of this attack in a paper that describes SubVirt, a malware hypervisor. Another infamous attack, called the Blue Pill, extended the SubVirt approach to create a permanent rootkit that could easily be launched on the fly using weaknesses in the factory-installed Windows operating system.

2.9.3 Static versus Dynamic Root of Trust

In most embedded systems, a chain of firmware must be executed to establish the secure initial state in which the TCB is up and running and controlling system security. Most commonly, the CPU first executes a small boot loader, burned into ROM at manufacturing time. Secure boot depends on a hardware-based root of trust; in this case, we depend on the fact that the ROM cannot be modified post-production. The ROM loader will often boot a more functional second-level boot loader residing in internal Flash. For example, many ARM-based embedded systems use the popular u-boot boot
loader. This boot loader will often boot the primary operating system or hypervisor that in turn boots its higher-level applications.

**Title-6**

The typical secure boot method is to verify the authenticity of each component in this boot chain. If any link in the chain is broken, the secure initial state is compromised. The first-stage ROM loader must also have a pre-burned cryptographic key used to verify the digital signature of the next-level boot loader. This key may be integrated into the ROM loader image itself, installed using a one-time programmable fuse, or stored in a local TPM that may provide enhanced tamper protection. The hardware root of trust must include this initial verification key. Chapter 4 describes the concepts of digital signature, usually implemented with public key cryptography.

The signature key is used to verify the authenticity of the second stage component in the boot chain. The known good signature must therefore also be stored in the hardware-protected area. The verification of the second-level component covers its executable image as well as the known good signature and signature verification key of the third stage, if any. The chain of verification can be indefinitely long. It is not uncommon for some sophisticated embedded and mobile computing systems to have surprisingly long chains or even trees of verified components that make up the TCB. Figure 2.31 depicts a sample three-level secure boot sequence. When the verification chain begins at system reset and includes all firmware that executes prior to the establishment of the runtime steady state, this is referred to as a static root of trust.
In contrast, a dynamic root of trust allows an already-running system (which may not be in a known secure state) to perform a measurement of the TCB chain and then partially reset the computer resources such that only this dynamic chain contributes to the secure initial state. Dynamic root of trust requires specialized hardware, such as Intel's Trusted Execution Technology (TXT), available on some (at the time of this writing, higher-end) embedded Intel Architecture-based chipsets. The primary impetus behind dynamic root of trust is to remove large boot-time components, which must run to initialize a computer, from the TCB.

Figure 2.31. Sample static root of trust secure boot sequence.
On Intel Architecture-based systems, the BIOS is often an extremely large piece of software that is used to initialize the system. Because it is a large, monolithic piece of software, the BIOS may (and has in some cases been proven to) contain vulnerabilities that can be exploited. By performing the dynamic reset (also sometimes referred to as late launch) after the BIOS has initialized the hardware, removing all privilege from the BIOS execution environment, the system in theory has reduced its TCB and improved the probability of a secure initial state. Unfortunately, several weaknesses, both in hardware and software, that implement the late launch mechanism, have been found by researchers, bringing into question the ability to achieve a high level of trust in complicated boot environments. Furthermore, while the Trusted Computing Group (TCG) has standardized TPM interfaces, implementations have yet to extend far beyond Intel Architecture-based computing environments.

### Key Point
The good news for secure boot is that most embedded and mobile computing systems rely on simple boot loaders that lend themselves well to the static root of trust approach that can be implemented without specialized hardware.

#### Title-7

Even a PC-based system can incorporate a custom, secure BIOS developed by embedded software security experts, if the embedded system’s security is worth that investment. The readers should consult a systems software vendor to understand the available options.

Another good example of the importance of secure boot is provided in the discussion of data-at-rest protection in Chapter 5.

#### 2.9.4 Remote Attestation
Secure boot provides embedded systems developers with confidence that the deployed product is resistant to low-level, boot-time firmware attacks. Nevertheless, a risk may persist in which sophisticated attackers can compromise the secure boot process. Furthermore, an attacker may be able to replace wholesale the deployed product with a malicious impersonation. For example, a smart meter can be ripped off the telephone pole and replaced with a rogue smart meter that looks the same but covertly sends private energy accounting information to a malicious website. Therefore, even with secure boot, users and administrators may require assurance that a deployed product is actively running the known-good TCB.

When embedded systems are connected to management networks, remote attestation can be used to provide this important security function. Once again, the TCG has standardized a mechanism for TCG-compliant systems to perform remote attestation using TPM-based measurements. Network access can be denied when a connecting client fails to provide proper attestation. Within TCG, this function is called Trusted Network Connect (TNC). However, a simple, hardware-independent approach can be used for any embedded system.

Let’s assume that the embedded system can communicate to the remote attestation server using a secure channel, such as IKE/IPsec or SSL (both discussed in Chapter 5). The initial session establishment uses public key cryptography. In particular, the static private key representing the identity of the remote embedded system is used to sign data that is then authenticated by the attester. As long as this private key and the client side of the secure connection protocol software are included in the TCB validated during secure boot, the attester has assurance that the embedded system is running some known-good firmware. Therefore, the mere act of a successful IKE or SSL session establishment can be used for remote attestation.
An improvement to this approach, providing assurance that the embedded system is running a specific set of trusted firmware components, is to have the client transmit the complete set of digital signatures corresponding to the TCB chain to the attester that stores the known good set of signatures locally. This is more difficult to implement because the signatures must be computed and saved at manufacturing time, before the embedded product is deployed to the field.

Title-8

2.10 Key Points
1. The operating system bears a tremendous burden in achieving safety and security.

2. The foundation of a MILS-based embedded system is the separation kernel, a small microkernel that implements a limited set of critical functional security policies, including data isolation, information flow control, damage limitation, and periods processing.

3. A separation kernel is considered a reference monitor when the kernel’s MILS policy enforcement mechanisms are always invoked, tamper-proof, and evaluable.

4. The MILS architecture requires the use of security-enforcing components whose functional requirements meet a high level of assurance.

5. Microkernel operating systems provide a better architecture for security than monolithic operating systems. 6. Memory protection is a fundamental requirement for robust embedded systems.

7. Virtual memory provides additional security features, including guard pages and location obfuscation, on top of basic memory protection.

8. System designers must plan for failures and employ fault recovery techniques.

9. Despite memory protection and virtual memory, malicious code can still take down a critical application by starving it of resources.

10. Partitioning, coupled with the ability to specify resource managers and multiple instance servers, enables system designers to obtain flexibility where needed and security where it is essential.

11. Device drivers are frequently the cause of system reliability problems and the target of hackers; thus, device drivers are some of the most important components of the operating system to isolate and protect.

12. Determinism is required to enforce secure time partitioning.

13. Many security problems are caused by poor access control architecture and/or implementation within the operating system or improper use of access control facilities by the embedded system designer.

14. The biggest reason why buffer overflows are so damaging is that insecure operating systems and software development techniques promote the use of ambient authority.

15. One of the biggest challenges embedded system designers face with respect to access control security policy is finding the proper balance between the granularity of policy and the maintainability and assurance of policy.
16. For most objects and resources, whitelists are preferable to blacklists due to the former’s tendency toward reduced privilege implementation and design.

17. A capability acts as the mechanism for access as well as the access right: there simply is no way for a subject to access an object if the subject does not possess the object capability.

18. Distributed capability systems point to a need for the operating system to provide a means by which capabilities can be confined within privilege domains as well as a means for revocation.

19. The new level of abstraction needed to cope with increasingly sophisticated, consolidated electronic systems is the operating system itself, not just the computer’s hardware resources.

20. The availability of system virtualization technology across a wide range of computing platforms provides developers and technologists with the ultimate open platform: the ability to run any flavor of operating system in any combination, creating an unprecedented flexibility for deployment and usage.

21. Because they can be no more secure than their underlying general-purpose host operating systems (which are well known to be vulnerable), Type-2 hypervisors are not suitable for mission-critical deployments and have historically been avoided in such environments.

22. The microkernel-based hypervisor, a Type-1 architecture, is designed specifically to provide robust separation between guest environments.

23. The key advantage to full virtualization over paravirtualization is the ability to use unmodified versions of guest operating systems that have a proven fielded pedigree and do not require the maintenance associated with custom modifications.

24. The addition of CPU hardware assists for system virtualization has been key to the practical application of hypervisors in embedded systems.

25. An often overlooked and undervalued virtualization capability in modern ARM micro-processors is ARM TrustZone.

26. It is important for developers to understand that use of a hypervisor does not imply highly assured isolation between virtual machines, no more than the use of an operating system with memory protection implies assured process isolation and overall system security.

27. One of the biggest impacts on security and efficiency in system virtualization is the approach to managing I/O across virtual machines.

28. A number of vulnerabilities (ways to circumvent protections) have been discovered in IOMMUs and must be worked around carefully with the assistance of a systems software/hypervisor supplier.

29. Remote device management is the answer to the hacker’s wildest dreams: the embedded system is imbued not only with Internet access, but also a means to remotely modify and patch software.

30. System virtualization can provide an effective solution to the problem of incorporating or retrofitting secure remote management to embedded systems: the legacy operating environment is uplifted into a virtual machine, securely isolated from remote management functions, such as connection authentication and configuration management, provided by a trustworthy hypervisor.
31. Embedded system developers must do all they can to ensure the trustworthiness of the hardware supply chain.

32. Beyond trusted hardware, we must ensure that the trusted firmware/software cannot be subverted during the boot process.

33. The good news for secure boot is that most embedded and mobile computing systems rely on simple boot loaders that lend themselves well to the static root of trust approach that can be implemented without specialized hardware.

Title-9

2.11 Bibliography and Notes


© 2012 Elsevier, Inc. All rights reserved. Printed with permission from Newnes, a division of Elsevier. Copyright 2012. For more information on this title and other similar books, please visit www.newnespress.com.

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, click here to sign up for the EDN on Systems Design newsletter.