VMXHAL: A Versatile Virtualization Framework for Embedded Systems

Lucian Mogoşanu * **Mihai Carabaş** * **Răzvan Deaconescu** *
Laura Gheorghe * **Valentin Gabriel Voiculescu** **

* *Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Splaiul Independentei nr. 313, Sector 6, Bucureşti, 060042 (e-mail: lucian.mogosanu, mihai.carabas, razvan.deaconescu, laura.gheorghe@cs.pub.ro)*
** *VirtualMetrix, Inc., 16738 Zumaque Street, PO Box 569, Rancho Santa Fe, CA 92067 (e-mail:gabi@virtualmetrix.com)*

Abstract: Given the continuous evolution of hardware capabilities for embedded systems, the idea of virtualization becomes practical. Virtualization on embedded systems provides the running of multiple types of applications on the same platform and reduction in hardware and power consumption costs through consolidation of existing functionality. We present a versatile virtualization framework, VMXHAL, that we designed, implemented and deployed on embedded systems, ranging from development boards to fully fledged hand-held devices. VMXHAL uses a reduced Trusted Computing Base of only 25,000 lines of code (compared to Xen's 150,000 lines of code), comprising a thin layer L4 microkernel and a minimal set of management applications. The thin layer design provides versatility allowing both baremetal (or native) applications and paravirtualized operating systems. As proof of concept, we built a virtualized dual-Android setup: two Android operating systems running simultaneously on a Galaxy Nexus phone, on top of our virtualization framework.

Keywords: Operating systems, Embedded systems, Virtualization, Microkernel, Security, Android.

1. INTRODUCTION

Embedded systems have known continuous advances in recent years, from small devices to fully fledged handsets. There has been constant development both in terms of hardware and applications. Current high-end embedded hardware such as System-on-Chips (SoCs) found in hand-held devices provide quad-core CPUs and gigabytes of RAM to the user. From a software perspective, Google Play Store gathers millions of applications designed for mobile devices.

Given the continuous evolution of hardware capabilities for embedded systems, the idea of virtualization becomes practical. Benefits of virtualization on embedded devices include reducing power and hardware costs through consolidation, preserving legacy code, enhancing security and availability through application isolation, as shown in (Intel Embedded Alliance, 2011). (Heiser, 2011) describes how virtualization use cases include running multiple operating systems and run-time environments on the same embedded system, securing communication devices, medical devices, automotive infotainment, smart home servers.

With the current state of capabilities for embedded systems, there are several projects out there that tackle the use of virtualization in the embedded world. (Heiser and Leslie, 2010), the Integrity Multivisor [1], (Lackorzynski et al., 2004), and (Hwang et al., 2008) use virtualization technology in order to run virtualized operating systems (mostly Linux and Android) on top of high-end embedded hardware.

However, current approaches provide little support for dynamic management and control of virtual machines (or domains) running on top of embedded hardware; where that is happening (Hwang et al., 2008), the hypervisor is rather large, meaning an extensive weight on the Trusted Computing Base. At the same time, though drivers represent most of the operating system code and are the cause of a large number of failures, existing solutions provide little support for isolating device drivers and I/O management functionality from other components, as described by (Ryzhyk et al., 2009).

We present VMXHAL, a versatile virtualization framework that we have designed, implemented and deployed on embedded systems, ranging from development boards to fully fledged hand-held devices. Our framework provides a thin virtualization layer of around 25,000 lines of code with dynamic management features and isolation of I/O device drivers.

VMXHAL is based on a minimal core, an L4 microkernel acting as a hypervisor, also providing support for baremetal (or native) applications. For dynamic control and domain management we designed a resource management mechanism, called *ResourceControl*, into the microkernel. We use it to implement *SecDom0*, a native

[1] `http://www.ghs.com/`

application that provides resource management policies, while retaining the goal of a minimal Trusted Computing Base. On top the L4 microkernel, we add *VirtOps*, an infrastructure based on the Linux kernel providing device driver virtualization and isolation without compromising critical system functionality. We use paravirtualization to enable Linux-based operating systems to run on top of the microkernel, ensuring support for platforms that offer limited hardware virtualization features.

As proof of concept, we built a virtualized dual-persona Android setup: two Android operating systems running simultaneously on a Galaxy Nexus phone, on top of our virtualization framework. Moreoever, the framework is deployed and runs on Pandaboard, using the same OMAP4 SoC as Galaxy Nexus, on Beagleboard, using OMAP3, and on the ARMv5 gumstix emulator. We are currently in the process of porting it on a Nexus 4 phone.

The contributions of this paper are as follows:

- We present a novel virtualization framework design that provides versatility through a thin virtualization layer, dynamic control features and isolation of I/O device drivers. It allows running both baremetal applications and paravirtualized operating systems.
- We describe *ResourceControl* a resource management mechanism that allows implementing *SecDom0*. *SecDom0* provides dynamic management of virtual machines (domains).
- We show how we use *VirtOps*, a Linux-based infrastructure, for providing I/O virtualization to domains running on top of VMXHAL.
- We present a dual-persona Android setup consisting of two Android operating systems running on top of the the framework and the challenges of getting them to run. The two Androids run on a Galaxy Nexus phone with current effort to bring them to a Nexus 4 phone.

The rest of the paper details the architecture, design principles, implementation details and evaluation of the framework. Section 2 briefs the most important concepts used throughout our work. Section 3 presents the architecture and design principles of the virtualization framework with Section 4 showing some of the technical challenges we have faced. Evaluation of the framework is shown in the form of the Dual Android phone in Section 5 and potential use cases in Section 6. We present similar approaches in Section 7 and provide final remarks in Section 8.

2. BACKGROUND

This section highlights the concepts required to describe our framework. In the first part we define microkernels and place them in the context of embedded systems. The second part discusses virtual machines and mechanisms used by microkernels to meet virtualization criteria.

2.1 Microkernels

Microkernels are a particular choice with respect to operating systems kernel design: they are built under the assumption that a kernel should only manage components that are absolutely needed to run a computer system. Microkernels also focus less on functionality and more on mechanisms, thus providing a separation between mechanism and policy. Usually they implement minimal functionality, comprising Inter-Process Communication (IPC) mechanisms, scheduling and abstractions for virtual memory and execution. This design philosophy is aimed towards reliability, separating critical components from the rest of the system.

While microkernels are more general-purpose than traditional Real-Time Operating Systems (RTOS), their minimal design approach makes them well-suited to run on modern embedded systems such as mobile phones or automotive platforms. Their versatility allows them to run dedicated tasks and general-purpose applications at the same time, while isolating the former from the latter.

Early examples of microkernels are given in (Hansen, 1970; Wulf et al., 1974; Cheriton, 1984; Accetta et al., 1986). Later microkernels, named second-generation microkernels, are described by (Tanenbaum et al., 1987) and (Liedtke, 1995). The L3 and L4 microkernels were built by Jochen Liedtke in the 1990s to illustrate that some of the limitations of earlier microkernels can be eliminated by design, the main problem addressed being performance.

Subsequent rewrites led to the so-called "L4 family": a number of derivatives designed on the same principles as L4, with the goals of performance and security in mind. We use an L4 microkernel to implement our framework. Other modern microkernels based on L4 are presented in Section 7.

2.2 Virtualization

Virtualization defines a software component called a *hypervisor*, that acts as an extended hardware abstraction for multiple virtual machines (VMs) and is responsible for management tasks such as virtual machine scheduling. Depending on the nature of the hypervisor, virtualization architectures are divided into two distinct types. *Type I* hypervisors, also called baremetal hypervisors, run on the top of the host, turning it into a dedicated virtualization environment. *Type II* hypervisors, known as hosted hypervisors, run as privileged components of a general-purpose operating system and often use the operating system's interface to achieve virtualization.

Microkernel-based hypervisors, dubbed microvisors (Heiser and Leslie, 2010), are type I hypervisors, relying on the microkernel interface to manage VMs. Since microkernels offer only a minimal set of mechanisms, they only virtualize a subset of the physical machine's hardware, namely processors and memory. Guest operating systems running on top of microkernels access I/O devices either by running most of the native drivers or by implementing a set of stub drivers that communicate with native microkernel applications or other virtual machines.

Most microvisors achieve isolation by implementing support for containers known as protection domains, as shown in (Ruocco, 2008; Lackorzynski and Warg, 2009). VMXHAL implements a similar mechanism called *secure domains* (SecDoms), providing isolation between guest operating systems as well as native applications. Secure access to kernel objects is often implemented by microkernels

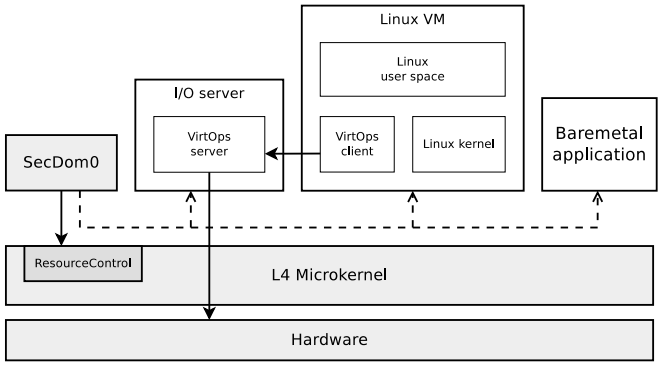

Fig. 1. Architectural diagram of the virtualization framework

using interfaces based on capabilities (Shapiro et al., 1999; Elkaduwe et al., 2008).

Given that hardware-assisted virtualization has yet to become mature on most embedded architectures (Mijat and Nightingale, 2014), microkernel-based hypervisors usually rely on *paravirtualization* to run guest operating systems: modify the guest operating system to replace privileged instructions with microkernel mechanisms. Guest user space applications run unmodified on the physical processor, trapping into the microkernel when privileged operations (synchronous or asynchronous interrupts) are executed. The microkernel then forwards the trap through an IPC to the guest kernel, which is modified to treat operations such as system calls and interrupts as IPC messages. Thus they are able to meet another criterion required for virtualization.

Finally, since most instructions on general-purpose embedded system architectures are unprivileged instructions, guest operating system performance is generally similar to that of its native counterpart. Many of the performance improvement efforts done in microvisors have achieved significant benefits (Wiggins et al., 2003).

3. ARCHITECTURE

The general architecture of our virtualization framework is outlined in Figure 1 and consists of the following:

- an *L4 microkernel* running in privileged mode
- a user space resource management application called *SecDom0*, implementing resource allocation policies and runtime management functionality for secure domains
- an *I/O server* secure domain, with direct hardware access capabilities, offering device driver functionality to other secure domains
- one or more *Virtual Machines*, particularly Linux-based operating systems
- one or more *baremetal applications*, running natively (without operating system support) on top of the L4 microkernel

We will describe the components mentioned above in the rest of this section.

The L4 microkernel acts as a hypervisor, providing minimal mechanisms such as thread scheduling, address space management, IPC or resource control. Isolation between components is achieved through SecDoms: each user space application (virtual machine or native application) runs in its own SecDom, being able to access only its own resources.

Microkernel resources are seen by user space entities as capabilities. While a particular capability is normally accessible only to threads inside a SecDom, the application designer can define inter-SecDom capabilities to implement communication with trusted drivers and other features such as virtual interrupts.

User space components are built using a set of L4 libraries providing access to the microkernel API. These are used by a more high-level interface that consists of a C library and a subset of the functionalities specified by the POSIX standard, containing for example the POSIX Threads library.

The microkernel only provides mechanisms for multiplexing access to the hardware resources, but there isn't any mechanism for explicit resource management. Questions such as "How much memory needs to be allocated for an operating system?" and "How many threads can exist in the system for a secure domain?" cannot be answered by the microkernel.

Resources are hard-coded at build time and can't be changed while the system is running, as the microkernel doesn't expose any interface to modify those values. As such, there is a need for a service running above the microkernel to take care of resource allocation for secure domains and for barebone applications running on top of microkernel.

3.1 SecDom0

In order to provide resource management, we propose a solution that creates a new special secure domain, called SecDom0. SecDom0 will be the only Secure Domain started at the beginning of the world [2]; the rest of the system will be controlled by this management component. The total available resources are granted to SecDom0, which will further allocate them to each new operating system instance on demand. This solution resembles that proposed by (Barham et al., 2003), with the difference that the management domain is much thinner in the implementation presented in this paper, as Xen uses a modified Linux kernel as Domain0 (this is their own appellation). The Xen hypervisor consists of 150,000 lines of code [3], while our virtualization layer comprises of 25,000 lines of code.

Before describing SecDom0, we list the resources provided by the microkernel. These resources are coupled to the hardware components. For example, a CPU is a resource that may not be partitioned, but a thread is a resource that enables CPU sharing.

In order for a secure domain (e.g. operating system instance) to run on top of the microkernel, the following resources are required:

- an initial address space

[2] beginning of the world - after the microkernel starts
[3] `http://wiki.xen.org/wiki/Xen_Overview`

- a capability list
- an initial thread to bootstrap the secure domain
- physical memory chunks/segments available

Providing the hardware abstractions presented above, SecDom0 can boot up any secure domain by allocating these resources, but particular steps need to be taken. The *SecDom0* action flow is as follows:

(1) An executable file in ELF format is generated by the programmer/user for a secure domain
(2) The ELF file is enriched with information about operating system resource requirements. Such information is the mechanism of interaction between any secure domain and SecDom0 when it comes to resource requests.
(3) The ELF file is uploaded on a storage device from where it is being read and interpreted using an ELF loader, developed in-house.
(4) Resources indicated in the enriched ELF file are allocated to the Secure Domain through an assignment process using the system calls provided by the microkernel.

In the end, the initial thread (called root thread) is started, becoming the entry point from where any guest operating system bootstraps itself. This process can be thought of as an analogy to a bootloader where it takes care of all the low-level initializations until the operating system can start booting on its own.

4. IMPLEMENTATION DETAILS

Section 3 presented a top-level view of VMXHAL, our virtualization framework. The lower parts of the architecture are the L4 microkernel, SecDom0 and native libraries that need to be glued up to work together. This section will extend implementation specific details for some of the basic native libraries, such as resource management and I/O virtualization. These form the basis of your virtualization framework. The development environment is described in subsection 4.2.

4.1 Resource Management

Designing SecDom0 revealed a deficiency in the microkernel mechanisms. There were no means through which one could create a new Secure Domain at runtime and bind to it a subset of the creator/parent's resources. Investigation lead us to the point where a new system call, which we called *ResourceControl*, was required.

Using the *ResourceControl* system call, resources of a Secure Domain can be altered and certain actions could open security breaches. If SecDom0 would create two new Secure Domains and would bind certain resources to each one, separation needs to be ensured: one Secure Domain couldn't access and control the other's resources, though these resources had initially been part of the same Secure Domain.

This separation is ensured by setting limits (base and size) for the capability list as a subset of the original list for each resource type. Creation of new Secure Domains can be done recursively; the system may end up with a hierarchical set of Secure Domains. Resource access rights are set up in such a way that any Secure Domain may access and modify any resources of his own, of its child Secure Domains or any other successor.

This feature allows one guest operating system to create as many Secure Domains as it wants (e.g. for each device driver), with no interaction with SecDom0 (its parent), thus ensuring a safe environment that can't be easily exploited. This no-interaction with SecDom0 supports a weakly coupled distributed system where SecDom0 is not a bottleneck; SecDom0 only deals with operating system boot up, not with the fine-grained management of guest requests.

Before the addition of the *ResourceControl* system call, resources for future Secure Domains had been statically configured through the build system. After initialization, the microkernel allocated necessary resources for all Secure Domains. Moving from static hard-coded allocation to dynamic management, where each Secure Domain may be independently controlled, comes with a cost of performance. The static allocation of resources at the beginning of the world (system startup) is done by the microkernel at system initialization without requiring a given system call to be invoked. Using dynamic management, SecDom0 runs in the user space of the microkernel and a system call is required for each resource-related operation, adding overhead.

Table 1. Secure Domain loading time – Static vs. Dynamic

SecDom	Static (ms)	Dynamic (ms)
Lightweight	1.3	22
Heavyweight	10	400

Table 1 provides a short overview of the time spent in creating a Secure Domain by the microkernel and then by SecDom0. While dynamic creation takes longer, it is still under 0.5 seconds even for the "heavyweight" SecDom. The lightweight Secure Domain was only printing a message to the console and had only 5 memory segments; the heavyweight SecDom was a Linux kernel and had about 30 memory segments. Time spent in allocating resources is growing faster when using SecDom0 (the slowdown factor ranging from 15x to 40x). This is explained by the fact that time spent allocating resources is proportional to the resource size. For example, in the case of memory mapping, granularity is 4096 bytes, and a larger segment needs more system calls.

4.2 Development Environment

Before continuing to the implementation of the virtualization framework components, we give a short description of the the supported hardware. The L4-based microkernel runs on ARMv5, ARMv6 and ARMv7-based hardware, more specifically XScale PXA255, Texas Instruments OMAP System-on-Chips (SoC) and Qualcomm Snapdragon SoCs. The latests SoCs supported are OMAP4460 and APQ8064, the former being implemented on Pandaboard and Samsung Galaxy Nexus phone, while the latter is part of LG Nexus 4 phone. The microkernel is highly portable contributing to the versatility of the VMXHAL framework.

To prove the usability of the framework we chose to paravirtualize a Linux kernel on top of the microkernel using the same technique as (Leslie et al., 2005a): creating a new `l4` architecture in the `arch/` directory. "Wombatized" kernel versions are 2.6.24, 2.6.29, 2.6.32 and 3.0.8. Initially we used simple filesystems with basic init programs and then moved to enhanced filesystems such as Ångström [4]. Our final objective was to run the Android framework on top of our paravirtualized Linux kernel, using a handset to demonstrate the versatility of our proposed solution. In the end we were able to run Android Ice Cream Sandwich (ICS) using the paravirtualized 3.0.8 Linux kernel on a Galaxy Nexus phone.

One of the issues with Android is its use of proprietary drivers for certain peripheral devices (e.g. GPU). Android ICS needs hardware graphics acceleration in order to boot-up, thus making the GPU one of the most important components in the handset.

Having only one virtualized operating system provides little practical advantages for our proposed solution; a dual persona phone was our future target as described in Section 5.

The L4 microkernel and SecDom0 provide CPU and memory virtualization. CPU virtualization uses threads as scheduling entities in the microkernel; there is a one-to-one correspondence between Secure Domain threads and microkernel threads. Memory virtualization is done through the use of address spaces for virtual memory. I/O virtualization is acquired by decoupling the driver backend for I/O devices in a dedicated Secure Domain; this Secure Domain hosts an I/O server that exposes a virtualization API which we dubbed VirtOps.

4.3 I/O Virtualization

In our goal for a Dual Persona Phone, we needed to virtualize at least storage devices (block), display, input (touchscreen) and networking.

The I/O device drivers require special treatment as they are not part of nor virtualized by the microkernel. Due to the microkernel requirement to be minimal, drivers run on top of the microkernel. I/O access to peripherals must be handled by the guest operating systems using the native drivers of the underlying hardware architecture. In the context of multiple guest operating systems, either a third party must control access to devices, or one of the guest operating systems must become a proxy for the others.

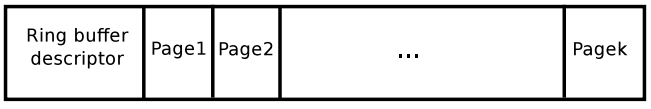

Fig. 2. Shared segment organization

Enabling guest operating systems to virtualize I/O devices in an extensible way directed us in developing a new library for virtual I/O operations, called VirtOps. The VirtOps provides an API for device virtualization using a client-server model. The client and server interfaces have to be generic (it's irrelevant what kind of data will be exchanged between the two sides) and have to ensure a safe and reliable data transfer.

In the rest of this section we will refer to a setup consisting of one I/O Server domain and one or more clients (*Linux VM*), as presented in Section 3. The I/O Server domain consists of a paravirtualized Linux distribution with exclusive access to all physical devices provided by the hardware platform. Its main function is multiplexing access of client operating systems to hardware in a secure way. For each client-server pair a shared memory segment is provided. This segment is used for data exchange in the form of a ring buffer. The shared memory segment is organized as shown in Figure 2. The first chunk represents the ring descriptor information and following are page-size aligned chunks used for effective data transfer.

Each client request is described by the format of `struct generic_ring_descr` element depicted in Listing 1 and a client descriptor holding specific data. The `shared_page_ind` field indicates the shared page used for data transfer.

Listing 1. Generic ring descriptor

```
struct generic_ring_descr {
    union {
      unsigned int raw;
      struct {
        unsigned int server_owned:1;
        unsigned int do_write:1;
        unsigned int server_err:1;
        unsigned int shared_page_ind:29;
      } X;
    } status;
    unsigned int size;
    unsigned int mem_offset;
};
```

The I/O Server maintains a list of registered local servers for different devices (e.g. timer, block IO, ethernet). In the registration process a server has to provide its type, a function to be called when a client request has arrived and one for registering new clients. Also, there is a list of registered clients from other domains, each having assigned a unique global identification number (GID). Similarly, a Linux Client has a list of local registered device clients.

On both sides (the I/O Server and the Linux client) there is a dedicated microkernel thread, called virtualization thread, used for dispatching server-client requests. Inter-domain requests are delivered using IPC messages, while data transfer is done through shared memory. This model is similar to the hardware model: a device places data in a memory location (our shared memory segment) and issues an interrupt (IPC in our model) to the driver to read information sent by the device.

In the next subsections we will briefly describe particular implementation details for the block, network, display and input virtualization. These are the main devices that need to be virtualized in order to have a usable operating system.

[4] `http://www.angstrom-distribution.org/`

4.4 Block I/O Virtualization

Block device driver paravirtualization required an image file for each client; this image file resides in a special secure partition of the I/O Server, inaccessible to untrusted user space utilities. The image file itself contains a partition table and a set of partitions, in order to provide a block device interface to a Linux client.

The core of the Linux block layer manages block I/O requests. Such a request is described by a `struct request` structure consisting of a set of segments, each of them corresponding to one in-memory buffer.

Each block I/O client request is sent to the I/O Server using VirtOps and there it is transposed relative to the image file content using the information provided by the `struct request`. The requested data blocks are loaded into the shared memory segment and then the client is notified to grab them. The infrastructure uses zero-copying enabling the client to use data directly from the shared memory segment, disabling unnecessary copying into its own private memory.

4.5 Network Virtualization

In addition to having access to external storage via the block device driver, another fundamental requirement for client secure domains is network connectivity.

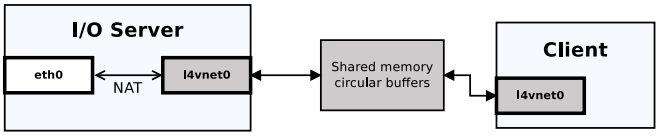

Fig. 3. Network virtualization architecture

Networking support is implemented in clients as an Ethernet driver called `l4vnet`. In each client, the virtual driver exposes an Ethernet interface to the operating system and user space applications. The virtualized `l4vnet` interface in each client can communicate to a corresponding interface in the I/O Server domain, by writing and reading Ethernet frames to/from circular buffers located in a shared memory segment, like shown in Figure 3. The I/O Server acts as a router, and performs NAT in order to enable Internet access for client secure domains.

For each secure domain, we can configure the driver in terms of number of network interfaces. Each client uses one interface, with the I/O Server having as many interfaces as there are clients. For each interface, we need to set the shared memory segment used for sending outgoing frames and the shared memory segment from which incoming frames are to be read.

In the current setup (in Figure 3), the network interfaces of the I/O server and the client are named `l4vnet0`. In the general case, where multiple clients would exist, the I/O Server interfaces are `l4vnet0` through `l4vnet<N-1>`, where N is the number of clients.

4.6 Display Virtualization

After enabling communication for the second Android (block I/O and network), we need to make it usable by

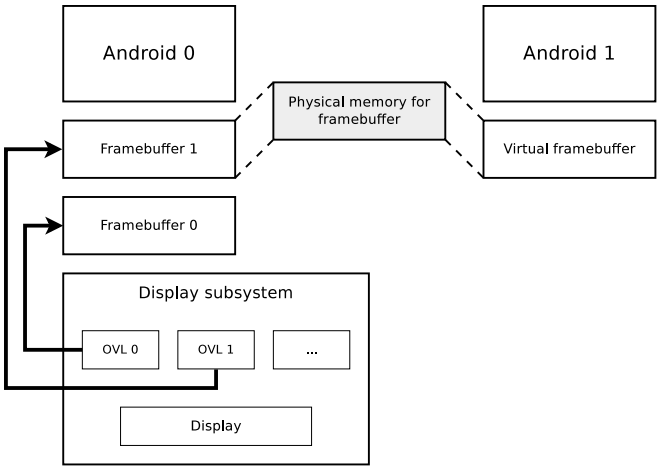

Fig. 4. Display virtualization

virtualizing the display (Carabas et al., 2014). As the second business oriented Android was not GPU intensive, we considered using software rendering. We disabled hardware rendering in the Android framework using the same method as the B Labs [5] Dual Android project. However, the B Labs demo uses VNC to virtualize display, leading to a larger performance overhead than our solution.

By default, the Android subsystem renders graphics in the first framebuffer, accessible via `/dev/graphics/fb0`. In the second Android, `fb0` is managed by a driver based on the virtual framebuffer implementation in the Linux kernel. The only difference is that, instead of being allocated from the kernel's memory, the virtual framebuffer is mapped in a shared memory segment defined in the build system of the L4 microkernel.

The display subsystem of the first Android also handles the screen of the second Android. We do this by allocating an extra framebuffer in Android0 and mapping it in the same shared memory segment as Android1's virtual framebuffer (see Figure 4).

The OMAP Display Subsystem (DSS) exposes a configuration interface in the Linux `sysfs` [6] interface. We use this interface to switch the display between the two Android instances. We make use of the *overlay* objects of the OMAP DSS. An overlay defines a rectangular area of a framebuffer and is used by an overlay manager to redraw only certain parts of the screen at a time and thus make the refresh process more efficient. In our case, we "borrow" one of the OMAP DSS's overlays and set it to contain the entire extra framebuffer. The switch between Android instances is done by enabling and disabling overlays via the `sysfs` entries.

4.7 Input Virtualization

In the Linux kernel, the Input subsystem is an abstraction layer between devices (keyboard, mouse, joystick, touchpad, and so on) and input handlers. The input devices capture input from user actions and produce input events that are dispatched to the interested handlers. We register multiple handlers (one for each client) for the same input

[5] `http://dev.b-labs.com`
[6] `https://www.kernel.org/doc/Documentation/arm/OMAP/DSS`

device (e.g. touchscreen), only one being active at a certain moment in time (the one that has the Android on the screen). In case of the client handler, events are dispatched through the VirtOps infrastructure to the input device driver of the Linux client.

5. EVALUATING VMXHAL: DUAL PERSONA PHONE

In the previous Section we presented the inner details of the VMXHAL framework, reasons for its versatility and steps made towards a practical demonstration using multiple instances of the Android operating system.

In today's business environment, an increasingly large number of employees use two phones: one for work and one for personal use (Brodkin, 2012). Gartner predicts that by 2017, half of the employers will require employees to bring their own devices to work [7], a policy dubbed BYOD – *Bring Your Own Device*. Through virtualization on embedded devices, one can provide mechanisms that allow running two operating system instances on the same phone, completely isolated from each other. Such a solution would enable employees to use a single dual-OS phone, relieving the need to carry two devices and providing flexibility in configuration and security.

The main advantage in having multiple operating systems running on top of a single device is related to the isolation degree. For example, having an operating system for a given environment provides a more secure environment than having only one OS instance for all environments.

The scenario stated above has been implemented with the VMXHAL virtualization framework described in this paper. We took a Galaxy Nexus phone and enhanced it with our L4 microkernel, the VMXHAL framework and two Androids on top of it: one Android for personal use and one for business use. One of the Androids had also taken the role of the I/O Server for device virtualization. This setup, described in Figure 5, is called a *Dual Persona Phone*.

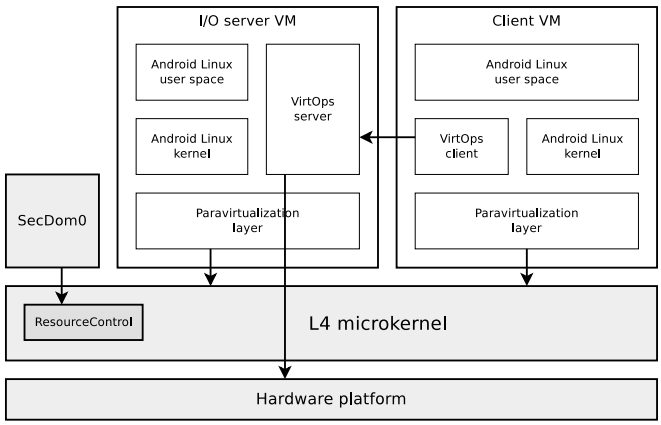

Fig. 5. Dual Persona Phone setup

The two Android VMs in our Dual Persona Phone are managed by SecDom0, which uses the ResourceControl mechanism to provide management functionality. VMs implement architecture-dependent operating system functionality (such as system call traps) using a paravirtualization layer. One of the Android VMs acts as an I/O server, having full access to hardware peripherals, while the other uses stub drivers to access devices. I/O virtualization is done on top of the VirtOps infrastructure described in Section 4.

During the process of bringing up a *Dual Persona Phone* using the Android framework, we encountered scalability problems. Android is a complex framework requiring a very large amount of memory: at least 256MB to boot-up and at least 512MB to be able to open the browser. The Galaxy Nexus phone has 1024MB of memory and only 768MB visible to user, the other chunk being left out for proprietary multimedia devices. Obviously running two Androids at the same time is at the limit. We needed to strip down the second Android and remove all unnecessary services in the startup process. In most of the cases this is enough as the business phone has reduced requirements, with software provided by the employee's company.

To make our solution more practical, we decided to move the implementation to a new device, Nexus 4, which has 2048MB of RAM. Currently under work, we obtained a port of the L4 microkernel on top of this platform.

6. FURTHER USE CASES

In this section we present other scenarios which could benefit from the usage of our virtualization framework. A possible application is represented by the emerging market of automotive In-Vehicle Infotainment solutions, where hardware costs would be significantly reduced by using virtualization. Another use case consists of the integration of virtualized operating systems with baremetal applications on Smart Home Server appliances.

6.1 In-Vehicle Infotainment

In-Vehicle Infotainment (IVI) is an umbrella term for hardware and software technologies employed in the automotive industry to provide entertainment functionality and other non-critical features such as automotive navigation. Various solutions have been proposed for infotainment systems based on Android (Chen et al., 2011; Macario et al., 2009; Al-Ani et al., 2010). IVI platforms include the architecture established by the GENIVI Alliance and Tizen IVI, which aims for GENIVI compliance itself (Ylinen, 2012). Another project directed towards IVI technologies is the Yocto Project layer for In-Vehicle Infotainment [8].

The main challenge of IVI is that of integrating a heterogeneous set of components which interact directly with the user: a music player, television, integrated GPS, a web browser, hardware interfaces (e.g. USB) and others. The complexity of such systems leads to specific requirements pertaining to security and performance. The system should be able to implement security policies where, for example, an application with Internet access wouldn't be able to access the current level of the gas meter.

[7] http://www.gartner.com/newsroom/id/2466615

[8] http://elinux.org/images/5/51/Yocto_Layer_for_In-Vehicle_Infotainment.pdf

Our solution can use the current Trusted Computing Base to implement arbitrary security policies. SecDoms can isolate secure and legacy components. Multiple Android and/or Linux instances could run virtualized on the top of the infotainment hardware platform to isolate front seat and back seat IVI. A further step toward this goal would involve certifying the L4 microkernel as a separation kernel (Rushby, 1981, 1989). Ensuring the reliability of the solution would be done by integrating High Availability features with the Infotainment applications.

6.2 Smart Home Server

In a similar fashion to IVI, the field of *Home Automation* illustrates ubiquitous computing by integrating so-called "smart" computing devices with household tools and activities, e.g. lighting, heating, appliances or entrance locks. Motivation for smart home ranges from energy efficiency to the increase of comfort for disabled persons. The design of home automation encompasses a variety of fields such as Artificial Intelligence, Systems Design and Human-Computer Interaction (Bravo et al., 2011).

We suggest the integration of various home automation components on a single hardware platform, with the help of virtualization features provided by the framework presented in this paper. The solution would consist of an Android/Linux virtual machine providing a user interface and legacy drivers, running in parallel with native applications used for dedicated functions, such as an Internet gateway or an electronic door locking mechanism. The architecture would meet the needs of dedicated applications (e.g. a RTOS or some other native application requiring full processor availability). It would also isolate the unreliable components from the rest of the system by design, ensuring that the overall system functionality remains uncompromised.

The main challenge in providing a solution for home automation (i.e. a *Smart Home Server*) based on our framework is that of providing a portable user space environment for baremetal applications. The current implementation provides support for a subset of the C library functionality and minimal POSIX. This proves to be insufficient for the needs of applications such as a dedicated stack for network routing. We aim to address this aspect in the future.

7. RELATED WORK

Similar approaches on virtualizing embedded systems either follow a thin layer microkernel-based design or a heavier hypervisor with extended features. VMXHAL provides features from both worlds, with a reduced Trusted Computer Base and security focus on one hand and a baremetal domain for dynamically managing other domains.

Projects similar to ours are based on the L4Ka::Pistachio and OKL4 (Heiser and Leslie, 2010) microkernels. Both L4 variants are able to run the Wombat kernel (Leslie et al., 2005b), a Linux kernel modified to run as a paravirtualized user-mode application. OKL4 aims towards the migration of its API to that of seL4 [9], a formally-verified L4 microkernel (Klein et al., 2009). With setup being done at build time, these L4-based designs lack a dedicated domain to dynamically manage virtual machines.

Similarly, (Lackorzynski et al., 2004; Lange et al., 2011) illustrate paravirtualized Linux kernels running on top of Fiasco.OC (Werner, 2012), a modern rewrite of the original L4 microkernel. L4Android has the same goal of providing a Dual Persona phone, its major disadvantage being that it doesn't support newer versions of Android such as Ice Cream Sandwich (Android 4.0), which require a working GPU driver to achieve a fully usable virtualized environment. The same is the case with the framework based on CODEZERO (Pham et al., 2013): the demos provided by B Labs [10] use VNC to provide display virtualization, which incurs an additional overhead caused by network transfer and frame compression, in contrast to our solution, based on framebuffer switching in the Linux kernel. Our Dual Persona Phone based on VMXHAL uses the framebuffer switching technique and provides improved user experience when compared to the B Labs work.

(Barr et al., 2010) provide a solution for mobile virtualization that uses a similar architecture to that of their desktop products. (Andrus et al., 2011) propose a framework using lightweight virtualization techniques to provide a Dual Persona phone based on a Linux kernel and the Android framework. Both solutions lack security by design. The first because it relies on hosted virtualization: attackers can compromise the integrity of the entire system if they compromise the host operating system. The second because it must include the Linux kernel in the Trusted Computing Base, and because it cannot achieve full isolation between the two virtual phones.

Xen, one of the more popular virtualization solutions, provides an ARM port [11]. While initially focused on ARM chips targeted at the server market, there is ongoing effort to support mobile devices (Hwang et al., 2008), even if chips provided no virtualization support. However, the Xen hypervisor possesses a relatively large Trusted Computing Base, with around 150,000 lines of code. Our L4 microkernel consists of a Trusted Computing Base of only around 25,000 lines of code, making it less prone to security vulnerabilities.

Green Hills [12] address security issues in embedded virtualization with their INTEGRITY RTOS. Its main advantages are its Separation Kernel Protection Profile (SKPP) and EAL6+ [13] Common Criteria certifications. Green Hills claims availability of their secure virtualization technology on hardware ranging from mobile devices to automotive though lack of publications and technical documentation makes it difficult to evaluate.

8. CONCLUSION AND FURTHER WORK

We presented a virtualization framework for embedded systems consisting of a thin microvisor, a separated domain for drivers and I/O management and dedicated do-

[9] http://os.inf.tu-dresden.de/pipermail/l4-hackers/2007/003149.html

[10] http://dev.b-labs.com/prebuilt-demos/dual-tuna-ics/

[11] http://www.xenproject.org/developers/teams/arm-hypervisor.html

[12] http://www.ghs.com/products/rtos/integrity.html

[13] Evaluation Assurance Level

main (*SecDom0*) for dynamic control of virtual machines. This design provides versatility, runnig on multiple devices and allowing the deployment of both baremetal applications and paravirtualized operating systems.

The framework extends existing embedded virtualization solutions such as OKL4 and L4Linux, being based on an L4 family microkernel and providing I/O isolation and dynamic management. It uses a rather small Trusted Computing Base, with about 25,000 line of code, compared to the 150,000 lines of code of the Xen hypervisor. The dynamic management component provided by *SecDom0* provides the ability to easily and fully configure and control non-privileged SecDoms and their resource allocation. With this design it is possible to restart a failing SecDom with no impact on other SecDoms, ensuring reliability.

We deployed the framework on several ARM-based devices, ranging from development boards to fully fledged hand-held devices. As proof of concept we built a Dual Persona phone setup, with two Android operating systems running simultaneously on top of the virtualization framework.

In the virtualization framework we implemented a new interface, VirtOps, to isolate drivers and ease development of I/O virtualization components. Isolating driver is important as they are a major cause of faults in operating systems.

We demonstrated the extensibility of the interface with a setup with two Android Linux making use of the storage, network, input and display devices.

Future work will focus on enhancing portability and use cases. Effort is under way to port the virtualization framework to a quad-core platform and providing full Symmetric Multi-Processing (SMP) support to the upper layer. New devices are to be targeted to ensure portability. We target deployment in commercial products and future applications such as automotive and infotainment, real time applications and baremetal native applications and drivers.

Porting the Linux kernel on Nexus 4 is work in progress as we also want to enhance the paravirtualization model based on Wombat (Leslie et al., 2005a). Our current model virtualizes space management operations (most notably page mapping/unmapping and cache flushing) using microkernel system calls. This incurs an overhead of up to 27% on system calls done by a simple command such as `ls`. The performance penalty is caused by frequent on-demand paging and cache flushing operations; we plan to eliminate these shortcomings by switching to a more robust virtualization design.

In order to enhance performance and portability with respect to applications and operating systems running on top of the framework, we aim to create a POSIX-like runtime environment running on top of the microkernel to be used by native applications, virtualized operating systems and drivers.

ACKNOWLEDGEMENTS

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

We would like to thank Gary Gibson and Val Popescu, of VirtualMetrix, Inc. for their continuous support and advice in our work. Gary has provided invaluable technical feedback and proposals and Val has always kept us on the path of practicality of our work. We thank our colleagues Petre Eftime and Cristi Condurache for their review and thoughtful remarks, and Andrei Buhaiu, Lucian Cojocar, Vali Priescu, Cătălin Moraru for their initial work and design decisions in the project.

REFERENCES

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. (1986). Mach: A new kernel foundation for unix development. In *USENIX 1986 Summer Technical Conference and Exhibition Conference; 1986.*, 93–112.

Al-Ani, T., Savarimuthu, T., and Purvis, M. (2010). *Android-based in-vehicle infotainment system (aivi).* Master's thesis, University of Otago.

Andrus, J., Dall, C., Hof, A.V., Laadan, O., and Nieh, J. (2011). Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 173–187. ACM, New York, NY, USA. doi:10.1145/2043556.2043574. URL `http://doi.acm.org/10.1145/2043556.2043574`.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5), 164–177.

Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., and Zoppis, B. (2010). The vmware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.*, 44(4), 124–135. doi:10.1145/1899928.1899945. URL `http://doi.acm.org/10.1145/1899928.1899945`.

Bravo, J., Fuentes, L., and de Ipiña, D.L. (2011). Theme issue:"ubiquitous computing and ambient intelligence". *Personal and Ubiquitous Computing*, 15(4), 315–316.

Brodkin, J. (2012). Ars readers confirm: We want dual-persona smartphones. `http://arstechnica.com/information-technology/2012/12/ars-readers-confirm-we-want-dual-persona-smartphones/`, [Last accessed on Oct 01, 2014].

Carabas, M., Mogoșanu, L., Deaconescu, R., Gheorghe, L., and Țăpuș, N. (2014). Lightweight display virtualization for mobile devices. In *International Workshop on Secure Internet of Things 2014*. IEEE.

Chen, M.C., Chen, J.L., and Chang, T.W. (2011). Android/osgi-based vehicular network management system. *Computer Communications*, 34(2), 169 – 183. doi:http://dx.doi.org/10.1016/j.comcom.2010.03.032. URL `http://www.sciencedirect.com/science/article/pii/S0140366410001647`. Special Issue: Open network service technologies and applications.

Cheriton, D.R. (1984). The v kernel: A software base for distributed systems. *Software, IEEE*, 1(2), 19–42.

Elkaduwe, D., Klein, G., and Elphinstone, K. (2008). Verified protection model of the seL4 microkernel. In *Verified Software: Theories, Tools, Experiments*, 99–114. Springer.

Hansen, P.B. (1970). The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4), 238–241.

Heiser, G. (2011). Virtualizing embedded systems: why bother? In *Proceedings of the 48th Design Automation Conference*, 901–905. ACM.

Heiser, G. and Leslie, B. (2010). The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, 19–24. ACM.

Hwang, J.Y., bum Suh, S., Heo, S.K., Park, C.J., Ryu, J.M., Park, S.Y., and Kim, C.R. (2008). Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, 257–261. doi:10.1109/ccnc08.2007.64.

Intel Embedded Alliance (2011). White paper: The benefits of virtualization for embedded systems. Technical report. URL `https://embedded.communities.intel.com/servlet/JiveServlet/previewBody/6956-102-1-2113/2011,%20Virtualization%20White%20Paper%20FINAL.pdf`. Last accessed on Mar 9, 2015.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, 207–220. ACM, New York, NY, USA. doi:10.1145/1629575.1629596. URL `http://doi.acm.org/10.1145/1629575.1629596`.

Lackorzynski, A. and Warg, A. (2009). Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, 25–30. ACM, New York, NY, USA. doi:10.1145/1519130.1519135. URL `http://doi.acm.org/10.1145/1519130.1519135`.

Lackorzynski, A. et al. (2004). L4linux porting optimizations. *Master's thesis, Technische Universitat Dresden*.

Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., and Peter, M. (2011). L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 39–50. ACM.

Leslie, B., van Schaik, C., and Heiser, G. (2005a). Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.conf.au*. Canberra.

Leslie, B., van Schaik, C., and Heiser, G. (2005b). Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.conf.au*. Canberra.

Liedtke, J. (1995). On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5), 237–250. doi:10.1145/224057.224075. URL `http://doi.acm.org/10.1145/224057.224075`.

Macario, G., Torchiano, M., and Violante, M. (2009). An in-vehicle infotainment software architecture based on google android. In *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on*, 257–260. IEEE.

Mijat, R. and Nightingale, A. (2014). Virtualization is coming to a platform near you. `mobile.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`, [Last accessed on May 24, 2014].

Pham, K.D., Jain, A.K., Cui, J., Fahmy, S.A., and Maskell, D.L. (2013). Microkernel hypervisor for a hybrid arm-fpga platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, 219–226. IEEE.

Ruocco, S. (2008). A real-time programmer's tour of general-purpose l4 microkernels. *EURASIP J. Embedded Syst.*, 2008, 11:1–11:14. doi:10.1155/2008/234710. URL `http://dx.doi.org/10.1155/2008/234710`.

Rushby, J. (1989). Kernels for safety. *Safe and Secure Computing Systems*, 210–220.

Rushby, J.M. (1981). Design and verification of secure systems. In *ACM SIGOPS Operating Systems Review*, volume 15, 12–21. ACM.

Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. (2009). Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, 275–288. ACM, New York, NY, USA. doi:10.1145/1519065.1519095. URL `http://doi.acm.org/10.1145/1519065.1519095`.

Shapiro, J.S., Smith, J.M., and Farber, D.J. (1999). *EROS: a fast capability system*, volume 33. ACM.

Tanenbaum, A.S., Woodhull, A.S., Tanenbaum, A.S., and Tanenbaum, A.S. (1987). *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ.

Werner, J. (2012). *Improving Virtualization Support in the Fiasco. OC Microkernel*. Master's thesis, Technische Universität Berlin.

Wiggins, A., Tuch, H., Uhlig, V., and Heiser, G. (2003). Implementation of fast address-space switching and tlb sharing on the strongarm processor. In A. Omondi and S. Sedukhin (eds.), *Advances in Computer Systems Architecture*, volume 2823 of *Lecture Notes in Computer Science*, 352–364. Springer Berlin Heidelberg. doi:10.1007/978-3-540-39864-6_28. URL `http://x.doi.org/10.1007/978-3-540-39864-6\_28`.

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. (1974). Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6), 337–345.

Ylinen, M. (2012). Tizen IVI Architecture.