Elliptic

Embedded Security you can Trust

Open Source Security Reference Platform For OmniShield User Guide

Document Number	Version
EDN-0839	1.0

Table of Contents

1 About this Guide 4
1.1 Audience 4
1.2 Assumptions and Limitations 4
1.3 Scope 4
1.4 Related Documentation 4
1.5 Acronyms 6
1.6 Terms 8
2 System Architecture 9
2.1 Base Privilege Levels 9
2.2 Base System Architecture 10
2.3 Base System Components 12
2.3.1 Hardware - OmniShield SOC MIPS32 Dual Core SMP Processor 12
2.3.2 Simulator 12
2.3.3 L4 µ-Kernel 12
2.3.4 L4 OS System Calls 12
2.3.5 Sigma-0 12
2.3.6 I/O Server 13
2.3.7 Ned (Init Process) 13
2.3.8 Moe (Root-Task) 13
2.3.9 High Bandwidth IPC-Calls 13
2.3.10 Karma VMMs 14
2.3.11 Hyper-Calls 14
2.3.12 Normal World Guest OS 14
2.3.13 Secure World Guest OS 14
2.4 Elliptic's Additional Components 14
2.4.1 Privilege Level 0 Root Kernel 16
2.4.2 Privilege Level 0 Hypervisor Microkernel 16
2.4.3 Privilege Level 1 Root User 16
2.4.4 Privilege Level 1 Hypervisor User-land 16
2.4.5 Privilege Level 2 Guest Kernel OS 16
2.4.6 Privilege Level 3 Guest User 16
2.5 Benefits of the OmniShield Architecture 17
3 Retrieving the Source Repo Packages 18
3.1 Package Retrieval Using Git 18
3.2 Environment Variables 18
4 How to Build the Demo 19
4.1 Prerequisites to Build Fiasco.OC 19
4.2 Build Fiasco.OC 19
4.3 Prerequisites to build SDK with Demo Hello World 22
4.4 Build Secure Kernel SDK with Demo Hello World 22
5 Secure Application Development 23
5.1 Basic Concepts of a Secure Application 23

5.2 Understanding the Application Programming Interfaces (API) 24
5.2.1 Normal World Client API 26
5.2.2 Normal World Client Configuration File 26
5.2.3 Secure World Trusted Applet API 27
5.3 Understanding the Distribution Structure 28
5.4 Normal World Application 29
5.4.1 Demo Hello World Application 29
5.5 Secure World Trusted Applet 32
5.5.1 Demo Trusted Applet 32
6 Shared Memory Driver Enhancements 35
6.1 Original Shared Memory Driver 35
6.2 Enhancements to the Shared Memory Driver 35
7 Paravirtualized Linux Character Driver 37
8 How to Obtain the Open Source Code for OmniShield 39
9 Limitations 40

1 About this Guide

The guide describes the following topics of interest:

- System Architecture Overview
- System Components
- Benefits of the Architecture
- How to build a Secure Application
- How to use the Normal World and Secure World API's

1.1 Audience

This guide is intended for someone that is implementing or considering implementing secure applications with message passing between the Normal World OS and the Secure World OS using Imagination's OmniShield enabled hardware (MIPS CPU) platform.

1.2 Assumptions and Limitations

This guide assumes that you are familiar with µ-kernels, virtualization of guest operating systems, memory management units (MMU), MIPS Architecture and family of processors, system on a chip (SOC) and the L4 µ-Kernel.

Currently the SOC software configuration is limited to 1 Normal World OS and 1 Secure World OS, the intended final product will be limited to 16 VMM ID's on one system.

1.3 Scope

The following topics are out of scope for this document:

- How to setup or use the Imperas Simulator
- How to setup or use Fiasco.OC environment
- How to setup or use Karma VMM environment

1.4 Related Documentation

For more information, consult the following documentation:

1. L4Re – L4 Runtime Environment l4e.org/doc/index.html
2. EuroSys 2006, Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies
3. Europa Hardware Architecture, version 01.00.184 not issued – live document, David Capon, 1 May 2014.
4. P5600 Multiprocessing System Datasheet, MD01024, version 01.00, IMG, 6 March 2014.
5. MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture, MD00082, version 5.03, IMG, 9 Sept. 2013.

6. MIPS Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, MD00086, version 5.03, IMG, 9 Sept. 2013.
7. MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture, MD00090, version 5.03, IMG, 9 Sept. 2013.

1.5 Acronyms

The following acronyms are used in this document:

ACRONYM	ACRONYM DESCRIPTION
API	Application Programming Interface
APPLET	A Small Application Executing in the Secure World
DCP	Digital Content Protection, LLC
HDCP	High-bandwidth Digital Content Protection System
L4 µ-Kernel	A minimal kernel (L4) to host virtual guest operating systems
L4Re	L4 Runtime Environment (a.k.a. Virtualized L4 OS)
NWA	Normal World Address
NWAH	Normal World Application Host
NW'ld	Normal World of User Level Programs
RNG	Random Number Generator
PRNG	Pseudo Random Number Generator
TRNG	True Random Number Generator
SKS	Secure Kernel Services
SRM	System Renewability Messages
SWA	Secure Word Address
SWAH	Secure World Application Host
SW'ld	Secure World of User Level Programs
SysCalls	System Calls
tAPP	Trusted Application
TAID	Trusted Application Identifier
TEE	Trusted Execution Environment
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

VMM	Virtual Memory Manager

Figure 1. Acronyms Table

1.6 Terms

The following terms are used in this document:

TERM	TERM DESCRIPTION
Client	Also known as the normal world application and draws parallels with the client-server architecture
Communication Channel	A channel in which the normal world and secure world communicate between the applications
Entangled	A method to make the normal world application highly dependent on the secure world application in such a way that makes the transactions between the two an inseparable binding between the normal world and secure world applications. There is a direct relationship between the entangled applications, the more dependent the normal world application is on the secure world application, the stronger the security of the application.
Rooted	An attacker has gained root access on the system and can modify or change system access privileges in the guest operating system.
Server	Also known as the secure world application and draws parallels with the client-server architecture
SKTEEC	Secure Kernel TEE Client. This is the client API that will be used to communicate to the secure application executing in the secure world.
TAID	The trusted application identifier is a value that is send to the secure world VM-to-VM Communication Proxy to route the application request messages to the secure world trusted application
µKernel	Micro Kernel, a very basic operating system with a small memory foot print, tuned for power efficiencies for lower power consumption of the device

Figure 2. Acronyms Table

2 System Architecture

This section of the document describes the system architecture that includes a new approach for security on the OmniShield platform. This is called virtualization. The L4 Micro Kernel is the base operating system in which two or more guest operating systems are managed. The guest operating systems are fully virtualized and only requires paravirtualized drivers for special and direct access to other system resources.

This fully virtualized environment introduces a new privilege system hierarchy that enforces security access inside the guest operating systems and externally in the Hypervisor Microkernel. Even if the guest operating system is "rooted" the privilege level only affects the guest OS where the root privilege is gained and executing. Any other guest OS's or the base L4 Micro Kernel are not affected and are protected from this type of unauthorized access. All guest OS's are protected by the base privilege levels enforced by the configured L4 Micro Kernel OS.

There are a number of base components that are required to be understood before developing a secure application. The following are the base system components:

1. Base Privilege Levels
2. Base Architecture
3. Base System Components
4. Elliptic's Additional Components

The benefits of this platform are briefly mentioned at the end of this section.

2.1 Base Privilege Levels

The table below has six (6) main privilege levels of access that range from 0 being the highest privilege level and 3 being the lowest privilege level. The following table provides a mapping between the privilege level and the role these privilege levels have in the overall system architecture.

Privilege Level	Role of Privilege Level
0 – Root Kernel	The L4 Kernel Privilege that manages hardware resources of the entire system. Access to physical devices requires permission at this level. The L4 Kernel requests are performed via the L4 OS SysCall interface.
0 – Hypervisor Microkernel	A hypervisor microkernel layer that supports Root User applications requests to the hypervisor via High-bandwidth IPC Call interface layer
1 – Root User	These are applications that provide system services to the Karma VM and Guest OSes. There are Sigma-0, I/O Server, Ned, and Moe as examples of root user processes. Inter-process communication is performed by the high bandwidth IPC Calls
1 – Hypervisor User-land	The hyper call layer that handles requests from a paravirtualized driver from either Guest OSes, which are fully virtualized Linux OS or the fully virtualized L4 OS
2 – Guest Kernel	This is the fully virtualized Guest OS kernel layer. An untouched Guest OS is not aware of the virtualization and only requires paravirtualized drivers to make requests to the Hypervisor User-land interface for special access to the system
3 – Guest User	This is a regular Guest OS user mode application. This application is managed by the Guest Kernel completely.

Figure 3. Privilege Level Table

2.2 Base System Architecture

There are three (3) basic areas of importance to understand. The first area (in green) is the normal world side, which comprises of a user mode (*Privilege Level 3 Guest User*) of a Linux Operating System, a Linux Operating System Kernel (*Privilege Level 2 Guest Kernel*) as the fully virtualized Guest Operating System Kernel, and the Karma virtual memory manager (*Privilege Level 1 Hypervisor User-land*).

The second area (in red) is the secure world Guest Operating System which is the L4 Runtime Environment (L4Re) as a user mode Guest User (*Privilege Level 3 Guest User*), L4 fully virtualized Guest Operating System Kernel (*Privilege Level 2 Guest Kernel*) and a Karma virtual memory manager (Privilege Level 1 Root User).

The third area (in blue) is the L4 Micro Kernel Operating system which hosts both normal world and secure world guest OSes. The blue area of the L4 Micro Kernel OS provides the physical access to hardware (*Privilege Level 0 Root Kernel*). At the same privilege level is the access to the Root Kernel via Hypervisor Microkernel.

These three (3) basic areas of importance, green is the *Normal World Guest OS*, red is the *Secure World Guest OS* and blue *L4 Micro Kernel Platform OS* make up the base system architecture six (6) privilege levels.

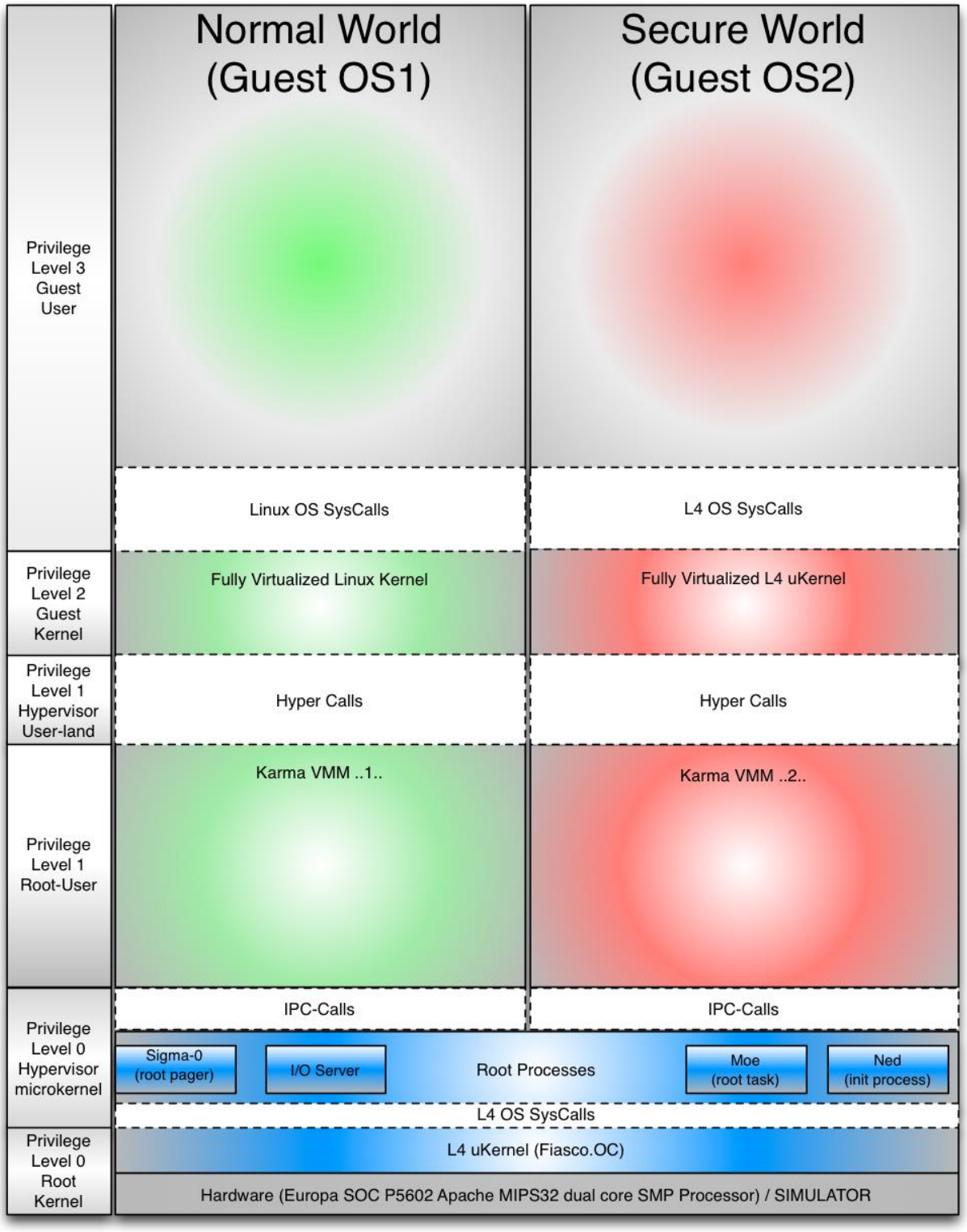

Figure 4. Base System Architecture

2.3 Base System Components

This section describes the system components and a description of their respective roles and responsibilities in the fully virtualized system as illustrated by the above Figure 1. Figure 4. Base System Architecture.

2.3.1 Hardware - OmniShield SOC MIPS32 Dual Core SMP Processor

OmniShield system on a chip (SOC) design implements the MIPS32r5 architecture. The MIPS32 architecture is based on a fixed-length; regularly encoded instruction set and uses a load/store data model. The architecture is streamlined to support optimized execution of high-level languages. Arithmetic and logic operations use a three-operand format, allowing compilers to optimize complex expressions formulation. Availability of 32 general-purpose registers enables compilers to further optimize code generation for performance by keeping frequently accessed data in registers.

The key hardware modules that provide virtualization support are the MIPS32 dual core SMP processors with virtualization modules and the PowerVR GPU) not shown in above diagram). The hypervisor microkernel runs in SMP mode. When a guest VM is started, the hypervisor microkernel assigns it one of 16 Virtual Machine IDs (VMIDs) that uniquely identifies the VM.

2.3.2 Simulator

OVPsim is free for evaluation and non-commercial use. It is compliant with SystemC for virtual platform behavior and peripheral components achieving IEEE 1666 compliance. It is released as closed source in a binary compiled form, and is maintained and supported by Imperas (www.imperas.com). OVPsim simulates the platform and provides a very flexible vehicle for embedded software development. OVPsim can simulate single-cpu, multi-cpu, or many-core platforms very efficiently. OVPsim uses Just in Time (JIT) code morphing, or binary translation, to achieve hundreds of millions of instructions per second simulation performance, and provides easy access to host workstation resources.

2.3.3 L4 µ-Kernel

Also known as the Fiasco.OC is a 3rd-generation µ-kernel (microkernel).

The Fiasco.OC kernel can be used to construct flexible systems. Fiasco.OC is the base for our trusted OS platform, which supports running real-time, time-sharing and virtualization applications concurrently on one computer. However, Fiasco.OC is both suitable for big and complex systems, but also for small, embedded applications. We have developed the L4 Runtime Environment, which provides the necessary infrastructure on top of Fiasco.OC for conveniently developing applications. Please refer to the L4 Features page for more information on the capabilities of Fiasco.OC.

Fiasco is a preemptive real-time kernel supporting hard priorities. It uses non-blocking synchronization for its kernel objects. This guarantees priority inheritance and makes sure that runnable high-priority processes never block waiting for lower-priority processes.

2.3.4 L4 OS System Calls

This is the L4 API layer for system calls to the L4 Microkernel (Fiasco.OC), any and all requests to gain access/control the hardware resources are requested via this API layer.

2.3.5 Sigma-0

Is a special server running on L4 because it is responsible of resolving page faults for the root task, the first useful task on L4Re. Sigma0 can be seen as part of the kernel, however it runs in unprivileged mode. Also, responsible for allocating, mapping and managing memory spaces for the root task and guest VMs,

i.e. mapping RVA to/from RPA entries.

2.3.6 I/O Server

Provides portable abstractions for iterating and accessing devices and their resources (IRQ's, IO Memory...), as well as delegating access to those resources to other applications (e.g., device drivers). Handles all platform devices and resources such as I/O memory, ports (on x86) and interrupts, and grants access to those to clients. Upon startup Io discovers all platform devices using available means on the system, e.g. on x86 the PCI bus is scanned and the ACPI subsystem initialized. Available I/O resource can also be configured via configuration scripts.

Io uses configuration can be considered as two parts:

a) the description of the real hardware

b) the description of virtual buses

Both descriptions represent hierarchical (tree) structure of device nodes. Where each device has a set of resources attached to it. And a device that has child devices can be considered a bus.

2.3.7 Ned (Init Process)

Ned's job is to bootstrap the system running on L4Re. The boot process is based on the execution of one or more Lua scripts that create communication channels (IPC gates), instantiate other L4Re objects, organize capabilities to these objects in sets, and start application processes with access to those objects (or based on those objects). For starting applications, Ned depends on the services of Moe, the Root-Task or another loader, which must provide data spaces and region maps. Ned also uses the 'rom' capability as source for Lua scripts and at least the 'l4re' binary (the runtime environment core) running in each application.

2.3.8 Moe (Root-Task)

This root task that is responsible for bootstrapping the system, and to provide basic resource management services to the applications on top. Therefore Moe provides L4Re resource management a multiplexing of services:

1. Memory in the form of memory allocators (L4Re::Mem_alloc, L4::Factory) and data spaces (L4Re::Dataspace)
2. CPU in the form of basic scheduler objects (L4::Scheduler)
3. VCON multiplexing for debug output (output only)
4. Virtual memory management for applications

Moe further provides an implementation of L4Re name spaces (L4Re::Namespace), which are for example used to provide a read only directory of all multi-boot modules. In the case of a boot loader, like grub that enables a VESA frame buffer, there is also a single instance of an L4Re graphics session (L4Re::Goos).

2.3.9 High Bandwidth IPC-Calls

A shared memory interface that shares buffers between VMs. This is the VM to VM communications

layer.

2.3.10 Karma VMMs

Contains and manages a supported guest or secure OS. There is one Karma VM per guest or secure OS. These VM's are just above the hypervisor and is considered a separate privilege level (Privilege Level 1) in the system as illustrated above architecture diagram.

2.3.11 Hyper-Calls

This is the application-programming interface to create hyper-calls that request services of the Guest/Secure OS's.

2.3.12 Normal World Guest OS

This is a fully virtualized Guest OS (e.g. Linux OS) to execute applications running on top of the Guest OS. This is to support the rapid development of applications that require a secure OS for security services provided by the Secure World OS.

2.3.13 Secure World Guest OS

This is a fully virtualized Guest OS (e.g. L4 OS) to execute as a Secure OS running applications that provide the normal world with entangled services.

2.4 Elliptic's Additional Components

The following diagram illustrates Elliptic additional contributed components (in red) that must be installed prior to testing or building the secure applications for this platform. This is partly in the "SK_SDK" distribution. The other parts are located in the OmniShield distribution.

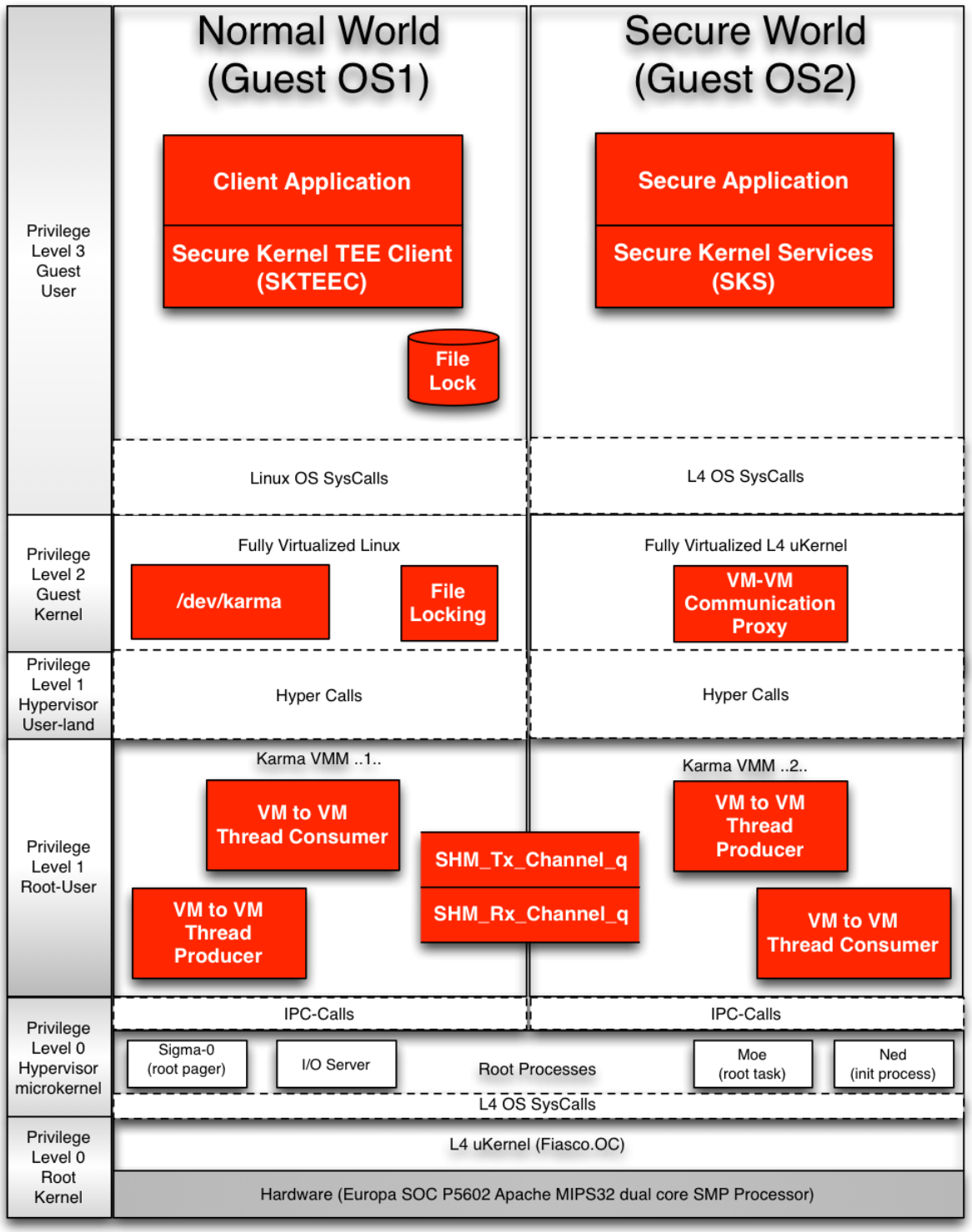

Figure 5. Elliptic Components (highlighted in red above)

2.4.1 Privilege Level 0 Root Kernel

The base L4 Micro Kernel (Fiasco.oc µKernel) as provided by the OmniShield distribution. There are no specific Elliptic components in this area.

2.4.2 Privilege Level 0 Hypervisor Microkernel

This privilege level is the interface to the Hypervisor Microkernel using IPC Calls to communicate with the Root User applications that provide access to system services or to hardware via L4 OS SysCalls. There are no specific Elliptic components in this area.

2.4.3 Privilege Level 1 Root User

The out of the box, standard Karma Virtual Memory Manager (VMM) does not have any facilities to communicate between guest operating systems. This drove the necessary enhancements of the shared memory driver (see 6 for more details). The shared memory driver uses IPC calls for the VM to VM communications. The parts of the shared memory driver are a thread consumer, thread producers, and a shared memory queue in common between each Karma VMM. One VM to VM Thread Consumer, VM to VM Thread Producer and access to the common shared memory queues for Tx and Rx data. This enhanced Karma VMM handles the memory management between the two or more guest operating systems. If memory from the virtualized Guest User (Privilege Level 3 Guest User) needs to be copied to the secure world application from the normal world, and vise-versa, it is managed at the (Privilege Level 1) Hypervisor User-land security layer. In other words by the shared memory driver for the VM to VM communications as illustrated above.

2.4.4 Privilege Level 1 Hypervisor User-land

This privilege level is the interface to the Karma virtual memory manager (VMM). This interface provides memory management services for the Guest OSes.

2.4.5 Privilege Level 2 Guest Kernel OS

The fully virtualized Guest Kernel (e.g. Linux Kernel) in the normal world and the fully virtualized Guest Kernel (e.g. L4 µKernel) in the secure world. In the normal world side there is a character device driver (/dev/karma) that can be accessed from (Privilege Level 3 Guest User) a user application. The device driver provides the mechanism for one or more application to send command requests to a secure world application. There is a file locking mechanism to ensure only one application in the normal world has control of the communication channel at a time. The file lock is held until the normal world application receives a response from the secure world application. Once the response is received the file lock is freed up for another normal world client application to create another normal world transaction to the secure world.

The VM to VM Communication Proxy, (a singleton kernel object) receives all commands and data from the normal world. It then invokes the secure world application and sends the secure world application the commands it received from the normal world application. The Secure world application receives the command and data then provides the response to the VM-VM Communication Proxy that will return the response to the normal world application holding the file lock.

2.4.6 Privilege Level 3 Guest User

On the normal world side, a client application will obtain and hold a file lock prior to sending a commands to a secure world application. After the responses are received from the secure world application, the client application will release the file lock for another client application in the normal world is able send commands to a secure world application. A number of client applications can queue up to obtain the file lock for secure world services.

2.5 Benefits of the OmniShield Architecture

The benefits of this architecture is the following list:

a) Rapid development of a user space application, in a popular OS, such as Linux, Windows, RTOS…etc

b) Rapid development of the Secure Application executing in the Secure OS part of the system.

c) Guest Operating Mode protects root assets by running a Guest OS and applications in a lower privilege level than the Hypervisor and the managing L4 Operating System.

d) Off target development

e) Off target testing and validation

3 Retrieving the Source Repo Packages

Before a building the secure application the supporting distributed open source packages are required to be installed on a hosted Linux build machine. Once all the packages are retrieved the environment variables and scripts need to be setup.

3.1 Package Retrieval Using Git

The main support packages that are required to build the secure application development are the following:

Package to Retrieve	Command to Retrieve Distributed Packages
Fiasco.OC	git clone https://sketch14@bitbucket.org/sanjayl/fiasco.oc-mips32.git
Karma VMM	git clone https://sketch14@bitbucket.org/sanjayl/karma-mips32-vz.git
Linux OS	git clone https://sketch14@bitbucket.org/sanjayl/karma-linux-mti.git
L4 Fiasco	See Fiasco.OC
Elliptic SDK	git clone $ELPSWROOT/securekernel/sk_sdk

3.2 Environment Variables

Environment Variable	Description of Use
ROOT_SDK	/home/user/sk_sdk
ELPSWROOT	/home/user/repository/cvsrep/sw_trees

4 How to Build the Demo

4.1 Prerequisites to Build Fiasco.OC

Prior to executing the build script, Install the Sorcery Code Bench Lite 2012.03.63 version 4.6.3 and any other cross compile tool chain required.

4.2 Build Fiasco.OC

The following is the build script called do_setup_build.sh. Execute this script with a parameter <any parameter> and the do_setup_build.sh will pull the repos using the git clone command. If no parameters are provided this script will just perform the build expecting the repos to already be on the local disk.

The following 10 lines setup the Cross-Compiler Tool Chain environment to compile the fiasco.git repo.

```
 1  Needs to be run from the parent directory of fiasco.git
 2
 3 set -e
 4
 5 . /tools/elpsoft/setgccpath.sh
 6
 7 . gse-mips 4.6.3
 8
 9 export CROSS_COMPILE=mips-linux-gnu-
10 export ARCH=mips
```

If there is a parameter in the command line then clone the following repos otherwise this script assumes it will perform the build commands.

```
11
12 if [ $# -eq 1 ]; then
13
14 git clone https://bitbucket.org/sketch14/fiasco.oc-mips32.git
fiasco.git
15
16 cd fiasco.git
17 git checkout elliptic
18 cd src/l4/pkg
19
26 git clone https://bitbucket.org/sketch14/karma-linux-mti.git
linux
27 cd linux
28 git checkout elliptic
29 cd ../../..
30 fi
```

This section copies the canned configuration files to the build direction in order to allow the build to run without any menu prompts.

```
 31
 32 cd fiasco.git
 33 cd src/kernel/fiasco
 34 make BUILDDIR=../../../build/fiasco
 35 cp src/templates/globalconfig.out.mips32-malta-vz-nomenu
../../../build/fiasco/globalconfig.out
 36 cd ../../../build/fiasco/
 37 make -j2
```

This section below copies the canned configuration files for the paravirtualized Fiasco Kernel.

```
 38
 39 cd ../../src/kernel/fiasco
 40 make BUILDDIR=../../../build-guest/fiasco
 41 cp src/templates/globalconfig.out.mips32-karma-nomenu
../../../build-guest/fiasco/globalconfig.out
 42 cd ../../../build-guest/fiasco/
 43 make -j2
```

Similar to the two previous section, this is setting environment variables for a canned configuration file that specifies the build options for the paravirtualized L4Re (Runtime Environment).

```
 44
 45 cd ../../src/l4
 46 make B=../../build-guest/l4
'DROPSCONF_DEFCONFIG=$(L4DIR)/mk/defconfig/config.guestos'
 47 mkdir ../../build-guest/l4/images
 48
 49 if [ ! -d ../../build-guest/l4/conf ]; then
 50   mkdir ../../build-guest/l4/conf
 51 fi
```

Building L4 as a guest operating system

```
 52
 53 #if [ ! -d ../../build/l4/conf ]; then
 54 #  mkdir -p ../../build/l4/conf
 55 #fi
 56
 57 cp conf/Makeconf_guest.boot ../../build-
guest/l4/conf/Makeconf.boot
 58 #cp conf/Makeconf_host.boot ../../build/l4/conf/Makeconf.boot
 59 cd ../../build-guest/l4
 60
 61 make -j2
 62
 63 make E=hello S=bootstrap
```

Building L4 kernel using the configuration files located in "conf" directory.

```
 64
 65 cd ../../src/l4
 66 make B=../../build/l4
'DROPSCONF_DEFCONFIG=$(L4DIR)/mk/defconfig/config.hostos'
 67 mkdir ../../build/l4/images
 68
 69 #cd ../../build/l4
 70 #make -j2
 71
 72 if [ ! -d ../../build/l4/conf ]; then
 73   mkdir -p ../../build/l4/conf
 74 fi
```

Setting up the configuration files and kicking off the build "make –j2" command to build the L4 guest OS and host OS for the fiasco.OC kernel.

```
 75
 76 cp conf/Makeconf_host.boot ../../build/l4/conf/Makeconf.boot
 77
 78 cd ../../build
 79 #make S=karma
 80 make
```

Setting up and building the L4 guest and host boot parameters. The karma hypervisor code is not part of the L4Re repository published by TU Dresden so the path for the karma code needs to be added to the configuration file using sed.

```
 81
 82 VAR_PATH=`readlink -f ../../src/l4/pkg/karma`
 83 sed -i "s@\"\/path\/to\/l4\/pkg\/karma\"@\"$VAR_PATH\"@"
.config
 84
 85 mkdir -p ../../build/linux
 86 cd ../../src/linux
 87 make O=../../build/linux karma_defconfig
 88 cd ../../build/linux
 89
 90 VAR_PATH=`readlink -f ../../src/l4/pkg/karma`
 91 sed -i "s@\"\/path\/to\/l4\/pkg\/karma\"@\"$VAR_PATH\"@"
.config
 92
 93 make
```

This is the last step of the build process. After-which an elf file should be available to execute.

```
94
95 cd ../../build/l4
96 make E=karma_fiasco_sys S=bootstrap
97
98
```

4.3 Prerequisites to build SDK with Demo Hello World

You have built the Fiasco.OC prior to this step

To acquire the SK_SDK portion of this project, please go to the GIT HUB location <TBD>.

4.4 Build Secure Kernel SDK with Demo Hello World

The following is the build commands for to building the SK_SDK repo. You should be in the repo directory that you have cloned as the SK_SDK.

```
1 SK_NO_LOG=1 TARGET=fiasco make demos

2 Then after running the Fiasco setup script go into
fiasco.git/build-guest/l4 and run

3 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK_IPC=1 make -C pkg/sk_ipc
clean && \
4 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK_IPC=1 make -C pkg/sk_ipc
&& \
5 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_hello
clean   && \
6 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_hello
&& \
7 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_olleh
clean   && \
8 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_olleh
&& \
9 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_base
clean && \
10 SKLIBDIR=/path/to/sk_sdk/pkg/ BUILD_SK=1 make -C pkg/sk_base
&& \
11 BUILD_SK=1 make E=sk_base S=bootstrap
12 Cd ./fiasco.git/build/l4
13 make E=karma_fiasco_sys S=bootstrap
```

5 Secure Application Development

The main goal of this platform to be able to develop secure applications quickly on a familiar platform such as Linux, Windows, L4 micro kernel, RTOS, or any other popular OS. Since this is a fully virtualized environment and can host a variety guest operating systems. These guest operating systems can be configured to be in the non-secure world or in the secure world. Picking a familiar operating system allows the rapid porting and development of client server type applications where the server component is in the secure world side of the platform. Since it is the platform that will enforce the access to the secure world, application developers can develop their client server applications off target until the application has been tested and verified. Once the application has been developed and tested it can then be easily ported with little effort to the OmniShield System on a Chip (SOC).

5.1 Basic Concepts of a Secure Application

The basic concept of this architecture is a client to server model. This relationship is already the standard operating model of all web browsers (clients) and web servers (servers). The client is considered untrusted and the server is considered trusted. All security related activities are performed on the server side also known as the secure world. The basic concept of the client-server model is to entangle the normal world application with the secure world application. This means the normal world application will not be able to function without the secure world application providing the essential entangled service.

The diagram (below, Figure 1. Figure 6) has identified three (3) basic components of creating a secure application in a client server model environment.

1. Client Application including secure kernel TEE client (SKTEEC) (green - Normal World Application)
2. Server Application including secure kernel services (SKS) (red - Secure World Application)
3. Communication Channel bi-directional between normal world to secure world
4. Same as #3

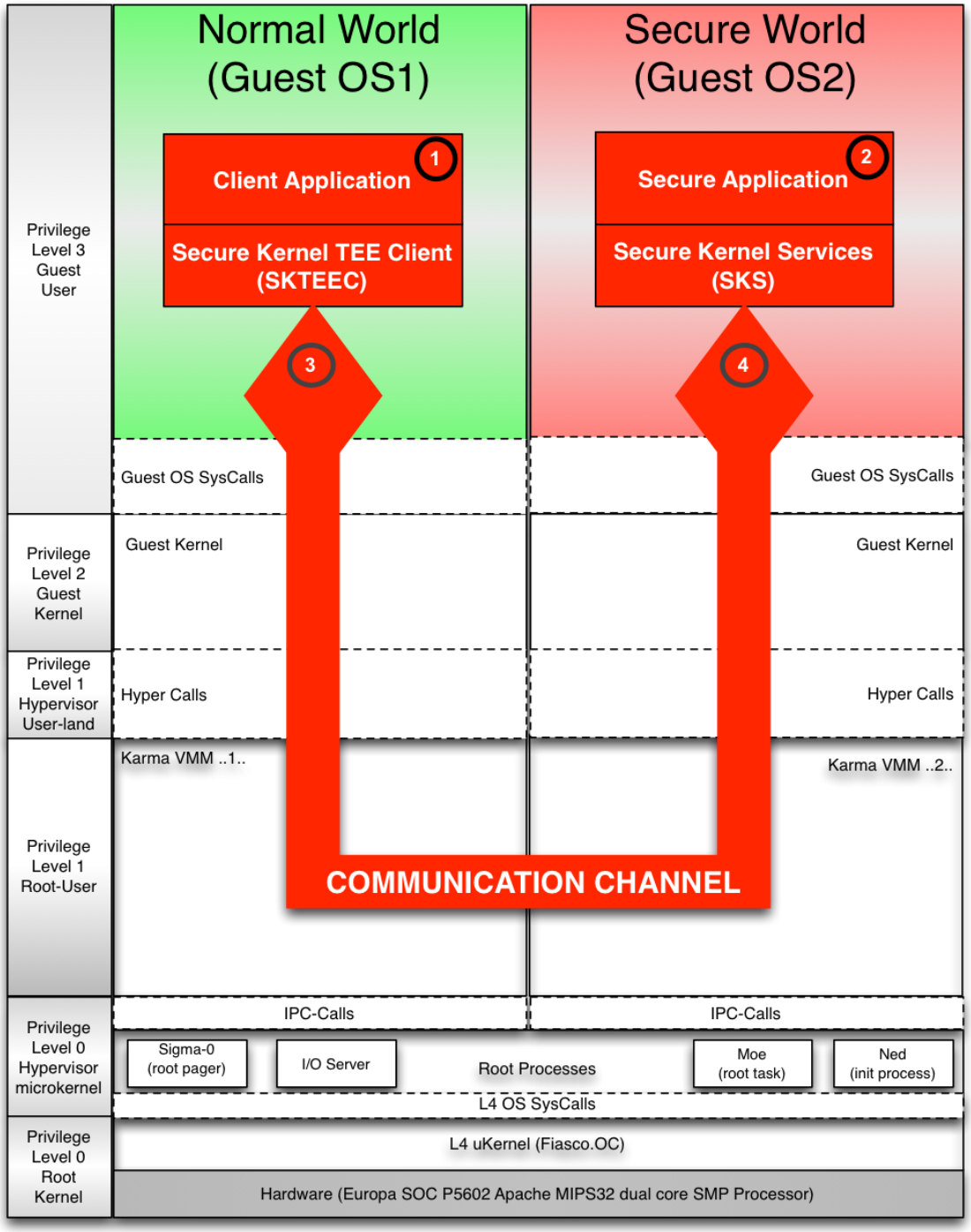

Figure 6. Three(3) basic concepts client, server and communication channel

5.2 Understanding the Application Programming Interfaces (API)

There are two main API's that need to be well understood prior to development of the application. One API is for the normal world client API (SKTEEC) to use to communicate with the secure world. The other API is for the secure world kernel services API (SKS) to receive the requests and return results of the command requests back to the normal world application. These API's are called the following:

- Normal World Client API - Secure Kernel TEE Client (SKTEEC)
- Secure World Service API - Secure Kernel Services (SKS)
- Platform API (SKPLATFORM) that is required on the normal world side for the platform specific communications between the normal world and secure world. These items are the following:
    - Normal World Platform API – Secure Kernel Driver Platform Initialization
    - Normal World Platform Object – Secure Kernel Driver Platform Initialization Parameters

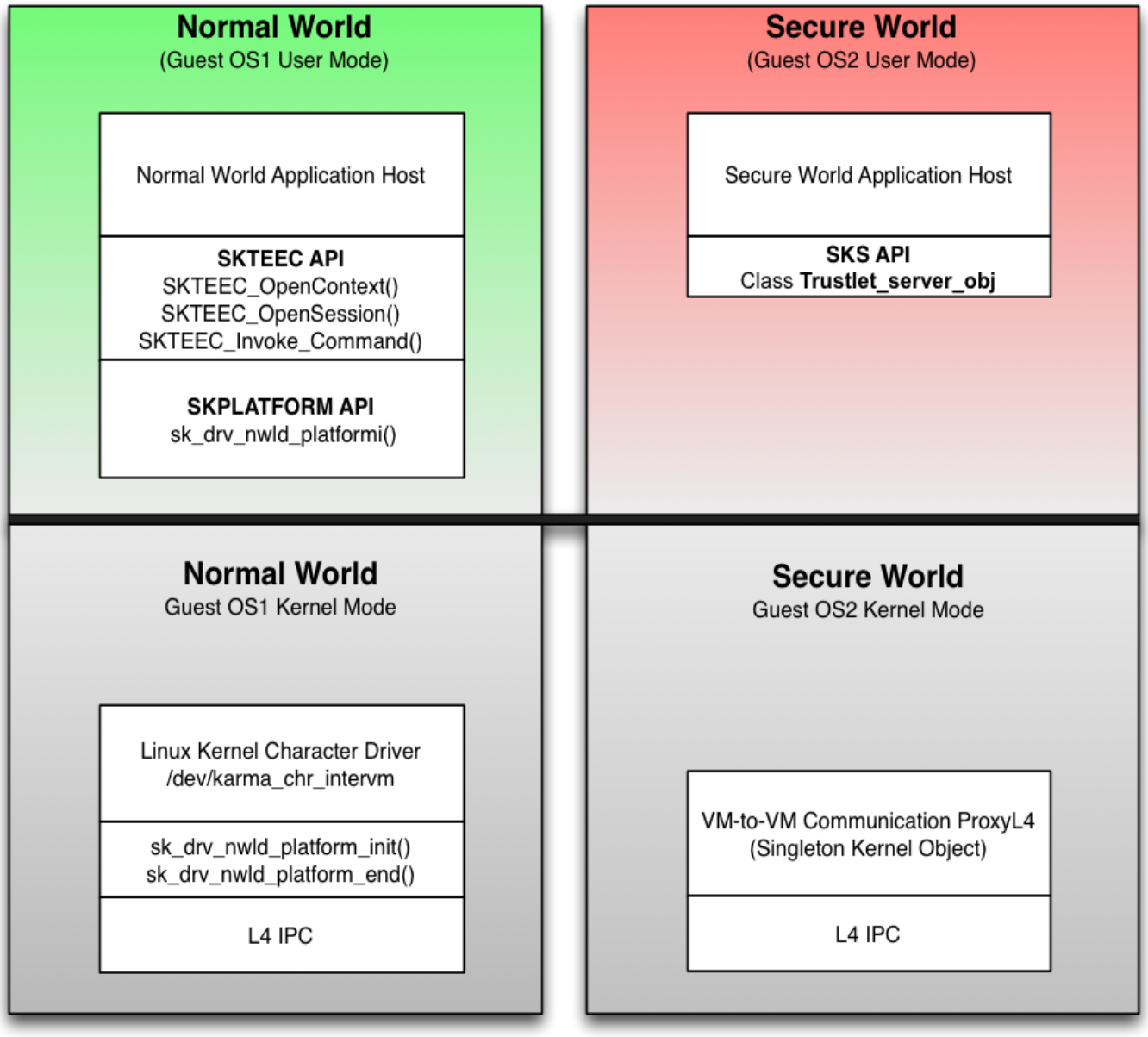

Figure 7. Understanding the Application Programming Interfaces

5.2.1 Normal World Client API

This is a user space API (*Privilege Level 3 Guest User*) located in $ROOT_SDK./src/nwld/include/sk_nwld_teec.h and is part of sk_sdk library that can be linked to applications that requires communication to the secure world. Please see the doxygen files in the sk_sdk repo for more detailed description of these APIs.

The following is a set of APIs that an application is expected to utilize to enable communications to the secure world:

Function Call	Description
SKTEEC_OpenContext(…)	Opens a context to the secure world
SKTEEC_CloseContext (…)	Closes a context to the secure world
SKTEEC_OpenSession(…)	Opens a session identifying the TAID to communicate with in the secure world.
SKTEEC_CloseSession(…)	Closes a session with a specific TAID application
SKTEEC_InvokeCommand(…)	Sends and Invokes the command to be serviced by the TAID application in the secure world
SKTEEC_AllocateSharedMemory(…)	Reserved for Future Use
SKTEEC_RegisterMemory(…)	Reserved for Future Use
SKTEEC_UnregisterMemory(…)	Reserved for Future Use

Figure 8. SKTEEC API Client Application Commands and Parameter

The following is a set of function calls expected to set up the platform driver for specific communications:

Function Call	Description
sk _nwld_platorm_init(…)	Opens and locks the communication channel exclusively until the resources are freed up.
sk_nwld_platform_end(…)	Closes and frees the resources allocated for the exclusive access of the communication channel.
sk_nwld_teec_platform	This is a context object that contains a data structure for communication channel management.

Figure 9. Normal World Platform Driver API

5.2.2 Normal World Client Configuration File

The following is an example configuration file to configure the TAID and parameters that provide limitations, modes and text labels to reference this application during run time.

```
//--------------------------------------------------------------------------
//
// The following file is used to configure the manifest for the
// hello world application
//
////////////////////////////////////////////////////////////////////////////
////

// 16 bytes (hexadecimal string)
[TAID]
12 01 02 03 04 05 06 07
08 09 0a 0b 0c 0d 0e 0f

// 1 byte:
[MULTI_INSTANCE]
1

// 1 byte:
[MULTI_SESSION]
1

// 4 bytes:
[MAX_HEAP_SIZE]
128

// 4 bytes:
[MAX_STACK_SIZE]
128

// String:
[SERVICE_NAME]
Elliptic Secure Kernel Application

// String:
[VENDOR_NAME]
Elliptic Technologies

// String:
[DECRIPTION]
Hello World
```

5.2.3 Secure World Trusted Applet API

This application programming interface handles the communication channel to the normal world application from the secure world side. Communication cannot be initiated from the secure world to the normal world, it must be initiated only from the normal world first. This is due to the secure application is always listening for incoming operations from the normal world. The normal world client application initiates the operations to be performed by the secure world and receives the results of the requested operation. This is a C++ class model where the Trustlet_server_obj class provides the SKTAPP_init_iface() for initialization when the applet is started and SKTAPP_recieve_operation_iface() when a normal world application initiates a command request to the secure world.

Class::Method() Calls	Description
Trustlet_server_obj::SKTAPP_init_iface(…)	Performs the initialization of the trusted application
Trustlet_server_obj::SKTAPP_receive_operation_iface(…)	Performs the receive operation of the trusted application from the normal world

Figure 10. Secure Kernel Services (SKS) API

5.3 Understanding the Distribution Structure

The structure of the distribution comprises of a ROOT SDK directory with sub-directories of documents (doc) and source code (src). The source code directory contains the common files (both), the normal world, the secure world, tools and platform specific sub-directories for specific platforms. Currently there is only one platform called fiasco. The high-level tree structure is illustrated in the table below.

```
ROOT_SDK_DIRECTORY
 cd
 ├── doc
 ├── libs
 └── src
     ├── both
     │   └── include
     ├── demos
     │   └── hello
     ├── nwld
     │   └── include
     ├── platform
     │   ├── fiasco
     │   │   ├── both
     │   │   │   └── include
     │   │   ├── nwld
     │   │   │   └── include
     │   │   └── swld
     │   │       └── include
     │   └── tcp
     │       └── include
     ├── swld
     │   └── include
     └── tools
```

Figure 11. SDK Directory Hierarchy

The following table below describes the individual directories that are included in the SDK:

Directory	Directory Description
ROOT_SDK	This is where the SDK is installed, a relative location to this directory.
./doc	Contains this document and other documents related to this SDK

./libs	Where the built libraries are located after the building the package. It should be noted this directory does not exist in the installed package. It is created during build time if it doesn't exist.
./src	The main source director where all the source files are located
./src/both	Used for common routines shared between normal world and secure world
./src/both/include	Include directory for common routines between normal world and secure world
./src/demos	Demo directory for demos included in this distribution
./src/demos/hello	Demo hello normal world application
./src/nwld	Off target normal world files
./src/nwld/include	Normal world include files
./src/platform	Platform specific files
./src/platform/fiasco	Fiasco platform specific files
./src/platform/fiasco/both	Fiasco platform specific for normal and secure world
./src/platform/fiasco/both/include	Fiasco platform specific for normal and secure world include files
./src/platform/fiasco/nwld	Fiasco normal world specific files
./src/platform/fiasco/nwld/include	Fiasco normal world specific include files
./src/platform/fiasco/swld	Fiasco secure world specific files
./src/platform/fiasco/swld/include	Fiasco secure world specific include files
./src/platform/tcp	Platform TCP communications
./src/swld	Off target secure world files
./src/swld/include	Off target secure world include files
./src/tools	Common tools for the SDK

Figure 12. SDK Directory Hierarchy Descriptions

5.4 Normal World Application

The normal world contains the fully virtualized Linux OS. This world is further reduced to two (2) separate operating modes: (1) User mode and the (2) Kernel mode.

The user mode Elliptic library contains the Secure Kernel TEEC Client API. The client (SKTEEC_) interfaces for the user space applications. It is a design and implementation option to link this library statically to the normal world application.

5.4.1 Demo Hello World Application

The hello_nwld.c is located in $ROOT_SDK/src/demos/hello directory of the sk_sdk repo. This is a demonstration of the SKTEEC and the sk_drv_nwld_platform APIs.

When developing your own application one of the first items you will have to do is include the “sk_nwld.h” and “sk_drv_nwld_platform.h” files. This will provide the prototypes and error codes for the API’s that will be used in the demo.

```
…
#include <sk_nwld.h>
#include <sk_drv_nwld_platform.h>
…
```

The next item is to create is a trusted application identifier (TAID). This is a 16 byte value that uniquely identifies the secure world application that will provide secure kernel services and a communication link to the normal world requesting application.

```
…
static SK_TAID taid[2] = {
 { 0x12, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
0x0d, 0x0e, 0x0f },
 { 0x13, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
0x0d, 0x0e, 0x0f },
};
…
```

In the hello_nwld.c main() function must define the “my_platform_param” and initialize this variable to the device name that will be listening for VM to VM communications. The device name is “/dev/karma_chr_intervm”. This is a Linux device driver operating in a virtualized kernel environment.

```
  …
 my_platform_param platform_param;
 sk_nwld_teec_platform obj;
…
```

The sk_drv_nwld_platform_init() initializes the defined item sk_nwld_teec_platform as well as open the character driver at “/dev/karma_chr_intervm”.

```
…
   platform_param.device_name = "/dev/karma_chr_intervm";
…
```

The SKTEEC_OpenContext() again uses the /dev/karma_chr_intervm” to create a context for the normal world application. Once the context is obtained, the SKTEEC_OpenSession is used to create a session between the normal world and secure world.

```
…
res = SKTEEC_OpenContext(&context, "/dev/karma_chr_intervm", 0x1010,
NULL);
if (res != 0) {
      printf("Failed to open a context\n");
      exit(EXIT_FAILURE);
}

res = SKTEEC_OpenSession(&context, &session, &taid[argc == 1 ? 0 : 1][0],
NULL);
if (res != 0) {
    printf("Failed to open a session\n");
    exit(EXIT_FAILURE);
}
…
```

Allocate memory for the SKTEEC_InvokeCommand() for the command and data to be sent to the secure world application.

```
…
operation.num_param                    = 2;
operation.param[0].meminfo             = calloc(1,sizeof(*operation.param[0].meminfo));

if (!operation.param[0].meminfo) {
      printf("Ran out of memory\n");
      exit(EXIT_FAILURE);
}
operation.param[0].meminfo->memory_size = sizeof(buf);
operation.param[0].meminfo->buffer      = buf;
operation.param[0].meminfo->flag        = SKTEEC_MEMREF_OUT;

operation.param[1].meminfo              = calloc(1,
sizeof(*operation.param[0].meminfo));
if (!operation.param[1].meminfo) {
    printf("Ran out of memory\n");
      exit(EXIT_FAILURE);
}
operation.param[1].meminfo->memory_size = sizeof(buf);
operation.param[1].meminfo->buffer      = buf;
operation.param[1].meminfo->flag        = SKTEEC_MEMREF_IN;
…
```

The demo application then performs the setup of the operations to be invoked by the normal world to the secure world. In this case the SKTEEC_InvokeCommand() API is used to send the command to the secure world and wait for the operation to complete in the secure world before returning from this invoke command API.

```
…
     sprintf(buf, "hello %d world", n++);
    res = SKTEEC_InvokeCommand(&session, 0, &operation);
    if (res != 0)
     {
          printf("Failed to invoke a command\n");
          exit(EXIT_FAILURE);
     }
    printf("Response(%d): [%s]\n", argc != 1, buf);
…
```

Once the secure world has processed the command it will send back the results and thus releasing control back to the normal world application.

The demo in an endless loop sends these commands counting the number of iterations.

Ideally the SKTEEC_CloseSession(), SKTEEC_CloseContext() and sk_drv_nwld_platform_end() would be called after all the invoke commands are performed and completed.

5.5 Secure World Trusted Applet

The secure world trusted applet contains the normal world virtualized L4 uKernel OS. This world is further reduced to two (2) separate operating modes: (1) User mode and the (2) Kernel mode.

The secure world framework classes are defined in the sk_base and sk_ipc directories and are included in the fiasco.OC GIT Repo in ./kyma_bitbucket_head/fiasco.git/src/l4/pkg directory. These form the base classes and framework of a trusted applet.

5.5.1 Demo Trusted Applet

There is a sk_hello which is the secure applet hello world that supports the normal world demo hello world.

Makefile

This is a standard fiasco.oc Makefile that contains a link to the trustlet_srv1.cc that contains the base classes and methods in sk_ipc and sk_base, that will support the two methods in the main.cc file. The ../../../sk_ipc/server/src/trustlet_serv1.cc must be in the SRC_CC of the make file as shown below.

```
PKGDIR          ?= ../..
L4DIR           ?= $(PKGDIR)/../..

SKLIBDIR ?= /tmp/sklib/
include ${SKLIBDIR}/incpaths
include ${SKLIBDIR}/swldlibs

CXXFLAGS += ${ELPSK_INCPATH}
LDFLAGS  += ${ELPSK_SWLDLIBS}

TARGET          = sk_hello
SRC_CC          = ../../../sk_ipc/server/src/trustlet_srv1.cc main.cc

include $(L4DIR)/mk/prog.mk
```

Trusted Applet

There are only one class and two methods when developing a trusted application in the secure world that are of importance. The Trustlet_server_obj is the class name and the two methods to implement are: SKTAPP_init_iface() and SKTAPP_receive_operation_iface().

The following code is the including of the header files that are standard and the sk_ipc header file that is required in order to include the trust application framework.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <signal.h>
#include <ctype.h>

#include <l4/sk_ipc/trustlet_srv1.h>
…
```

The following code shows the one class Trustlet_server_obj and the two methods that are required to be implemented in order to initialize the trusted application and to respond to the normal world application requesting the operation.

```
…

int Trustlet_server_obj::SKTAPP_receive_operation_iface(l4_umword_t cmd,
sktapp_operation *op)
{
   int   i;
   char *r, *s;

   r = (char*)op->param[0].mem.addr;
   s = (char*)op->param[1].mem.addr;
   for (i = 0; i < op->param[0].mem.memory_size; i++) {
      s[i] = toupper(r[i]);
   }
   return 0;
}

int Trustlet_server_obj::SKTAPP_init_iface()
{
   printf("SK HELLO TRUSTLET -- INIT\n");
   return 0;
}
```

6 Shared Memory Driver Enhancements

This section describes the enhancements to the original inadequate the “Shared Memory Driver” that was provided in the core distribution. It also illustrates how much effort and work has been provided by Elliptic Technologies in achieving higher performance and improved stability of the core distribution. The following sections describe the original shared memory driver and the bug fixes, refactoring and enhancements made to the shared memory driver.

6.1 Original Shared Memory Driver

The original shared memory driver in the L4 µKernel blocked entire emulation system during operations and was only half duplex when communicating between Virtual Machine’s hosting the guest operating systems. This was only an example demonstration of how to implement a shared memory driver for the L4 µ-Kernel. This was not an adequate example to base any work that requires high bandwidth of data between the Guest OS and the Secure OS. Also it should be noted there were a few bugs in the demo that caused instability to the emulator.

6.2 Enhancements to the Shared Memory Driver

This memory driver doesn’t exist in the L4 µKernel distribution (reference location here), this is a major enhancement to the core distribution of the L4 µ-Kernel. The following features and enhancements are included in this distribution, they are:

Shared Memory Driver Features/Enhancements	Description of the Feature/Enhancements
New Shared Memory Driver	Created a new shared memory driver
Supports Full Duplex	Allows the Guest OS and the Secure OS to communicate without blocking each other.
Supports Multithreaded Transactions	Internally manages buffers using a multithreaded approach for more asynchronous performance.
Supports High Speed Buffer Copies	Uses MIPS32 registers to perform high-speed buffer copies.
Supports Blocks on Buffer Reads	Allows applications to block on a buffer read until it is filled or times out.
Supports Non-Blocking Buffer Writes	Does not block the application writing data to buffer.
Supports Big Buffer Packets up to 4K	Allows the Guest OS Application to allocate buffers up to 4K between the Guest OS and the Secure OS and vise-versa.
Supports Variable Chunk Sizes for Managed Buffers	Internal optimization for managing buffers of variable chunk sizes
Supports for Bare Metal Applications	Allows Bare Metal Apps to call into the hypervisor for service.
Added Critical Sections and Mutex Support	Internal optimization for critical sections and blocking on Mutexes.
Added Thread signaling	Internal optimization for signaling threads to wake up and service their buffer FIFO queues.
Added Hyper-call bidirectional memory management	Allows for hyper-calls to request for a specific size of buffer without blocking until a buffer is available.
Added Clear Error Messages to failure scenarios	Created the error handling and reporting code to failure scenarios.
Created Test Bench Framework	A method to add more test cases

Created Tests for Test Bench Framework	A bunch of tests to test the performance and stability of the shared memory driver
Created Test for Round trip between Guest OS and Secure OS	Some specific test cases for round trip messaging and signaling.

Figure 13. Shared Memory Enhancements

7 Paravirtualized Linux Character Driver

This section describes the work that was involved in developing a character driver to send/receive messages to/from the guest OS- Linux, from the user application that initiated the sending of these types of messages to/from the Secure OS – L4, a trusted application that receives and responds to the commands it processes.

This is a para-virtualized Linux interface to the Elliptic inter-virtual machine communication system is via a standard Linux character driver interface. The driver implements the POSIX files system API for `open(), close(), read(), and write()`. The device number is hard coded to 254. A line was appended to the rcS script in the root file system image to create the device node for the driver. See the example below for this additional line.

mknod -m 660 /dev/karma_chr_intervm c 254 0

added Linux device module to

busybox-1.22.1/_install/etc/init.d/rcS

The operation of the driver is tightly coupled to the karma hypervisor. In most present day inter-virtual machine communication used with secure operating systems during the course of a transaction. While the hypervisor blocks the entire virtual machine (VM), including the Guest OS and above User space layer until the transaction is completed on the Secure OS and a releasing response message triggers the blocked VM to process the queued message that it was waiting for.

In the Elliptic system only the process making the `read()` call is blocked. During initialization of the normal world virtual machine the hypervisor adds a virtual IRQ to the GIC of the normal world's virtual machine (VM). When a process in the Guest OS Linux makes a `read()` call to the driver, the read() passes the number of bytes requested by the process to the hypervisor via a bidirectional hypercall. As there are no established synchronization primitives such as semaphores and mutexes to shared resources.

A more performant shared resource manager of a virtual machine and a hypervisor are tightly coupled and coordinated in our demo. This type of sharing algorithm still needs to be developed in order to avoid race conditions and dead locks between the paravirtualized inter-virtual machine communication driver and the hypervisor's shared memory devices.

The hypercall `karma_shmem_ds_set_read_size()` requests a set number of bytes from the shared memory device. It queries the number of bytes currently available, and requests an interrupt when the number of requested bytes are available if they are not initially available. It was necessary to make this hypercall an atomic operation in order to avoid the race a condition where the driver queries the number of bytes available in the shared memory device and finds the number of bytes are less than the number requested by the caller, more data arrives in the shared memory device increasing the number of bytes above the number requested, the character device blocks and never receives the interrupt. This mechanism was also used for performance reasons.

In a normal character driver attached to hardware the hardware would simply trigger an interrupt every time new data was available. In this system this would involve a context switch between the hypervisor and the virtual machine (VM) so if the hypervisor only triggers an interrupt when the requested number of bytes are available the context switching over head is avoided.

These are valuable insights into making this technique work seamlessly in the OmniShield platform using the para-virtualized Linux character driver. As mentioned in the limitations section of this document a block driver will have more flexibility in accessing user data and addressing responses and allowing for multiple-

simultaneous-bidirectional behavior and a less of a need for blocking the entire virtual machine (VM).

8 How to Obtain the Open Source Code for OmniShield

For instructions on how to obtain the L4 microkernel please visit the following website:

http://os.inf.tu-dresden.de/fiasco/doc.html

Download instructions:

http://l4re.org/download.html

9 Limitations

Currently there are limitations that should be noted. The following list identifies the limitation and a brief description of the limitation. These limitations are not necessary the final design or product but a description the current state of the system.

Limitation	Description of Limitation
VM Instances Supported	One (1) Guest OS and one (1) Secure OS
VM types supported	Linux and L4 (Bare-metal is deprecated)
Shared Memory Driver – Buffer Copies	5 Copies of Buffers from Guest OS to internal VMM buffers and again copies VM buffers to the Secure OS application buffers.
Shared Memory Driver – Threads	Supports only 2 threads per Guest OS VMM
Unsigned Trusted Apps	Should be validated on disk and in memory on the Secure OS
Encrypted Trusted Apps	Should be encrypted on disk and decrypted into the Secure OS
Unsigned and Encrypted Apps	Contains both an encrypted signed image on disk and should be decrypted and verified in memory in the Secure OS.
Paravirtualized Linux Driver	Hard Coded 254 Device Number
TAIDS	Are fixed numbers

Figure 14. Limitation Table