Public Key Accelerator: Driver Design and Implementation

Architecture Document

Rev 2.3

Document Number: MLNX-15-XXXX

Check into portalx to obtain the doc number!!!!!

www.mellanox.com

Contents

List of Figures

List of Tables

Revision

Table 1: Revision

Rev	Date	Author	Change Description

Purpose

This document provides a description of the design and implementation of the Public Key hardware accelerator software, also referred to as PKA stack. The software manages and controls the EIP-154 Public Key Infrastructure Engine, a FIPS 140-3[1] compliant Public Key Accelerator (PKA) and operating as a co-processor to offload the Host processor.

Scope

This document presents the rationale behind the design of PKA software, and specifies the design decisions that guide the implementation of the user API, the PK library and the Linux device driver.

Definitions/Abbreviation

Table 2: Abbreviation

Definitions/Abbreviation	Description
IPsec	Internet Protocol Security
SSL	Secure Sockets Layer
TLS	Transport Layer Security

Related Documentation

This section lists the Related Documentation.

Table 3: Related Documentation

Document Title	Description

[1] http://csrc.nist.gov/groups/ST/FIPS140_3/documents/FIPS_140-3%20Final_Draft_2007.pdf

Document Title	Description

1 Design Rationale

The Public Key Accelerator (PKA) offloads the Host processor, it provides high performance computation of several complex arithmetic operations, e.g., modular inversion, modular exponentiation, and high-level operations such as DSA/ECDSA generation and verification. The EIP-154 Public Key Infrastructure Engine includes a sub-module for True Random Number generation, as well. These operations are useful for a wide range of security applications. The EIP-154 can assist with SSL acceleration (Public Key operations) or a secure high-performance Public Key signature generator/checker.

1.1 Requirements and assumptions

The PKA software stack should be designed as a complete framework for hardware acceleration; in other words, it should implement an API for standard support such as kernel crypto API and OpenSSL. On the other hand, the API might be used to implement user applications requiring Public Key acceleration.

The PKA software stack is expected to run on the BlueField (BF) controller. The controller consists of an ARM cortex-A72 (AArch64) and executes a Linux distribution. Thus, the PKA software will be used to interface to the Public Key Infrastructure (PKI).

It is intended to allow multiple instances to control and share PKI resources; the framework should implement mechanisms to isolate dedicated resources and synchronize accesses to the shared resources.

1.1.1 OpenSSL

The OpenSSL[2] library provides an open source implementation of the SSL/TLS protocols. From the point of view of cryptographic operations, OpenSSL is based on a synchronous blocking programming model.

The PKA API design aims to provide an interface to accelerate applications based on OpenSSL library. Applications should be able to process asynchronous OpenSSL cryptographic operations on dedicated hardware. Current release defines ENGINE cryptographic module support to enable asynchronous OpenSSL operations, i.e., non-blocking approach that supports a parallel-processing model (OpenSSL ENGINE, 2016).

ENGINE objects are typically used to support specialized hardware. These objects act as containers for implementations of cryptographic algorithms, and support a reference-counted mechanism to allow them to be dynamically loaded in and out of the running application.

ENGINE objects have two levels of reference-counting to match the way in which the objects are used: structural reference and functional reference. Structural reference allows the

[2] http://www.openssl.org/

ENGINE to be initialized elsewhere than typical environments. A functional reference allows usage of ENGINE's functionality. Functional reference can be obtained from an existing structural reference to the required ENGINE. PKA API implementation should implement plugins to use a reference to ENGINE in order to provide crypto acceleration.

1.1.2 IPsec

In this document, we will focus on IPsec and its implementation within the native Linux cryptographic API, also known as the *scatterlist crypto API*. There are actually two APIs provided: one for user transforms, *transform API*, and one for registering algorithms, *algorithm API*. Note that transform API is general purpose; not only used by IPsec but it also provides transform services for other subsystems, such as encrypted filesystems, strong filesystem integrity, the random character device (/dev/random), network filesystem security (e.g., CIFS) and other kernel networking services requiring cryptography (Lumpkin & Phillips, 2006)

CAUTION:

OpenBSD Cryptographic Framework (OCF) and Asynchronous Crypto Layer (Acrypto) non-native APIs are not within the scope of this document (upstream issues). Note that OCF and Acrypto are designed specifically to accommodate asynchronous cryptographic hardware, attaching hardware is a matter of following the documentation and examples for hardware support that is already included.

The PK API tends to support asynchronous PK operations enabled by native Linux cryptographic API. In order to access the hardware, a Linux device driver should be implemented. The driver will then ensure the offload of crypto operations issued by Linux kernel services.

NOTE:

There are existing APIs to add asynchronous device support to the native API; These project, tools and APIs are currently under investigation;

Cryptodev-linux project: http://cryptodev-linux.org/

Crypto Engine Framework: http://www.linaro.org/blog/core-dump/crypto-engine-framework/

kernel Crypto API architecture: http://www.chronox.de/crypto-API/

1.1.3 Hardware resources

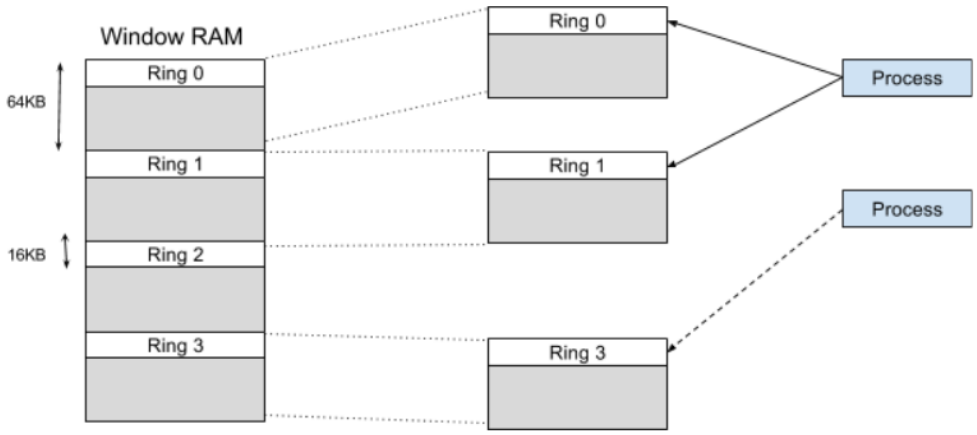
From the point of view of the controller, the PKI consists of a set of control status registers and memory pools:

1.1.3.1 Window RAM

A contiguous 64KB memory block partitioned into 4 regions each with a command descriptor ring and a result descriptor ring, and data vectors. Command descriptor ring and result descriptor ring can be used either as one ‘overlapping’ ring or as two separate ‘non-overlapping’ rings. In the following, the ring pair and its related data memory pool are referred to as a *Ring*. The current hardware defines 4 regions of 16KB each, which may be co-located or placed at different locations in window RAM. Each region contains a command descriptor ring, a result descriptor ring and data vectors memory pool. Descriptors contain pointers to the vector data within the 16KB region.

The PKA driver is responsible for partitioning the window RAM. It has to allocate memory for command/result descriptor rings and data vectors. Generally, data memory is larger than descriptor rings memory.

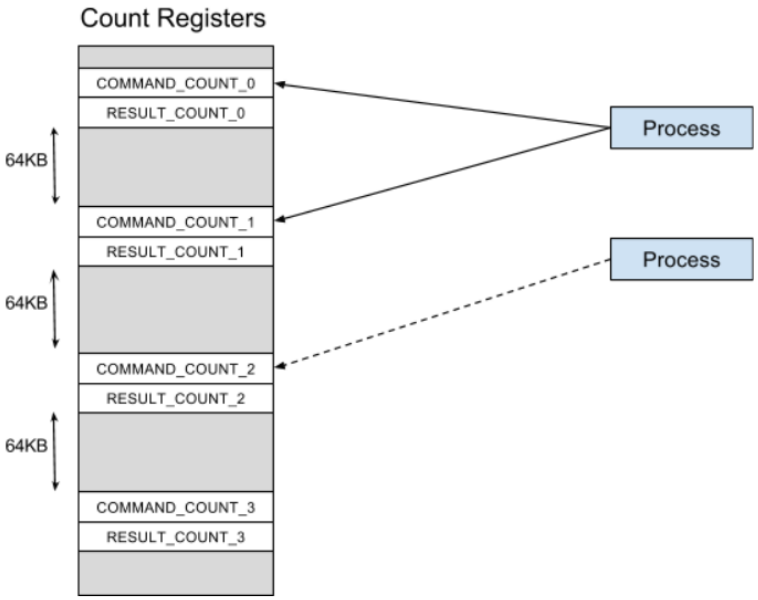
Figure 1 Window RAM



1.1.3.2 Ring Counters

There are two hardware counters per Ring, the command count register and the result count register. These counters have side-effects and trigger the command processing, i.e., enables the EIP-154 master firmware, when commands have been written to a given command descriptor ring or results have been read from the related result descriptor ring.

Figure 2 Count Registers



1.1.3.3 Ring information control/status words

The EIP-154 master controller use some words of buffer RAM to store read/write pointers and statistics for the rings, providing progress indication and a re-sync capability. These words are kept locally by the driver.

The PKA driver writes the ring base addresses, ring size and type, and initialize (clear) the read and write pointers and statistics. Upon a PKA command, the master firmware will read and partially update the information words.

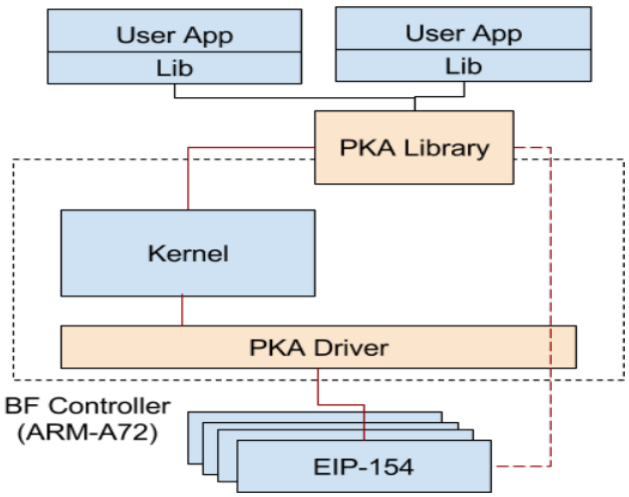
1.2 Use cases

NOTE:

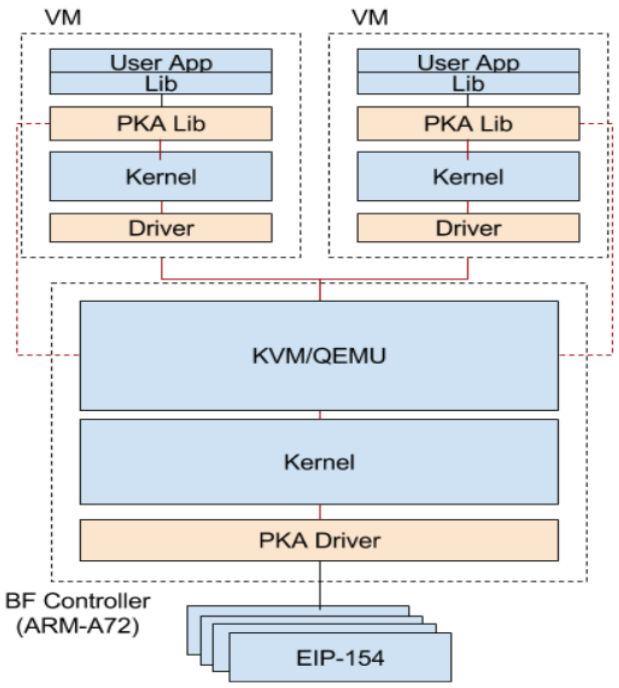
Use cases presented below give an overview of the expected work as well as long term vision to extend this work. One of the main purpose of this work is to attempt to provide a complete implementation to cover the following use cases (short and long term objective).

Also note that currently the document presents PKA driver implementation for ARM-based applications and support some features to enable virtualization.

1.2.1 ARM-based applications



1.2.2 Virtualization



NOTE:

Bare-metal hypervisors are not within the scope of the document. Software tools provided by Mellanox Software team include an edited Linux distribution (created with Yocto) and KVM/QEMU hypervisor layer.

2 Design Decisions

The PKA framework consists of separate but related component parts; an API, a library and a device driver. The driver provides kernel support and manages hardware resources, the library offers direct access to the PKI and handles PK commands, and the API specifies interfaces that are used by third-party modules and makes available a bunch of public key operations.

From software perspective, PKI resources are viewed as a set of devices which are isolatable from all other devices in the system. Since Rings are used to interface to the EIP-154 master firmware controller, those might be allocated to different instances.

NOTE:

We define an instance as crypto context that is using a set of PKI resources and that actively issues PK comands. In the following, we use ‘instance’ or ‘crypto instance’ to refer to a third-party code that uses the API or the library services.

2.1 PKA model

At startup, the PKA device driver initializes the hardware by booting-up the EIP-154 (i.e., loading the firmware and setting interrupts), partitioning the window RAM and writing the applicable ring information at start of the buffer RAM. After that, the EIP-154 master firmware is operational and it will be able to process PK commands. When an instance requests PKI resources, the library verifies whether Rings are available then assign Rings as needed. It performs a memory mapping of the corresponding Rings, configures the command descriptor ring according to Ring information control/status words, and writes the input data vector. Once a result is ready, it reads the result descriptor ring and copies the output data vector (if applicable). Result status may be reported to clients using polling mode.

From this illustrative model, we can see that PKA library controls Ring allocation. Once a given instance owns a Ring, it won’t be interfered with by another instance, i.e., allocated Rings cannot be used by another instance.

2.1.1 Device Initialization

During the initialization phase, the driver configures the AIC (Advanced Interrupt Controller) so that all interrupts are properly recognized. It must set the signal polarity (High Level/Rising edge) for each individual interrupts using AIC Polarity Control Register (AIC_POL_CTRL), and must set the signal type for each individual interrupt using AIC Type Control Register (AIC_TYPE_CTRL).

The boot-up sequence of the EIP-154 requires three firmware images: Boot image loaded into PKA_MASTER_PROG_RAM, Farm image loaded into PKA_BUFFER_RAM or into PKA_SECURE_RAM whether High Assurance mode or non-High Assurance mode is needed, and Master image loaded into PKA_MASTER_PROG_RAM.

Finally, the driver partition the window RAM with different regions and writes the applicable ring information at start of the PKA_BUFFER_RAM. A complete and detailed boot-up sequence is described in the INSIDE Secure's document (Firmware Reference Manual, 2013).

This phase requires specific knowledge of firmware location, direct access to PKA RAMs and should be executed a single time during system startup. Thus, these operations are executed purely by the device driver within controller operating system, at module initialization. We assume that kernel module for device driver is a loadable module, thus customers may not need PK I/O block.

2.1.2 Device Assignment

As noted in the previous sections, ring information control/status words, ring counters and window RAM are referred by Ring. The library code is responsible for allocating dedicated Rings to crypto instances.

The device driver controls and manages at most 4 PKA I/O blocks. Each device has 4 Rings, i.e., a pair of command/result descriptor rings within 16KB window RAM and 2 count registers. The driver organizes the PK resources, first by shared resources which refer to an I/O block, then by dedicated resources which refer to Rings. The total of dedicated resources is as the number of Rings by mean 16. Each Ring is accessed through its file descriptor.

The device driver creates device files for each Ring. The library call can request a Ring by simply opening its corresponding file. When a given instance requests resources, the driver finds free Rings, marks it as unavailable, and returns it. It looks trivial but PKA library may do one or multiple *open()* calls to find free Rings which can be used by the client, and return its related data. Conventional *RR* algorithm (modulus-based algorithm) is used to load balance among PKA I/O blocks then among its set of Rings. Once a Ring is no more used by a given instance, it must be released so it is marked again as available.

In order to support direct access to allocated resources, the library code calls *mmap()* on the Rings file descriptor returned by *open()* call, to map directly-accessible hardware words and registers. The driver performs a translation of virtual address space to physical address space, then returns a page for the requested resources. Note that one or more pages are needed to allow an instance accessing Rings, i.e., *mmap()* is called to map ring counters and related window RAM region.

2.1.3 Command Processing

After the device initialization and device assignment phases, the PKA is ready to receive commands. The library implements function to configure command descriptor rings by setting command type, operand offset and operand size. Upon the command count register is incremented, the command processing starts and the result count register is decremented. The

crypto instance can retrieve the result descriptor within the result ring and copy the associated output vector(s).

Interfaces are provided to submit PK operations at user space level and kernel level. The APIs wrap up client requests and offer direct access to resources from user process to enable high performance hardware acceleration. The PKA library is responsible for writing and reading command result descriptors, and for appending descriptors to command/result descriptor rings. Moreover, an instance can run over one or more threads. The APIs support one queue per thread, each queue is attached to one or multiple rings. At this level, there is no need to define priorities per thread nor per operation.

Polling mode is used to report result status to the associated instance. Indeed it allows the process to execute alternative tasks and avoids blocking loops. On the other hand, interrupts are enabled to inform kernel services which offloads PK operations to the EIP-154 that a given result is ready.

From this description, different APIs might be defined:

User API, to wrap up client requests and submit PK commands (asynchronous call); It allows to process client requests, configure command descriptors, and retrieve results;

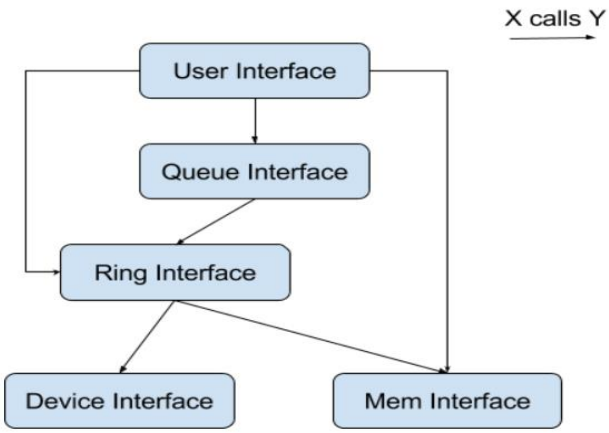
Queue Management API, to deal with client requests within multithreading environment and leverage hardware limitations;

Ring Management API, to manage command/result descriptors within hardware descriptor ring and to communicate with the hardware controlled by the EIP-154 master firmware;

Memory Management API, to allocate/free memory for input/output data vectors;

Device Management API: to abstract and manage PK hardware resources.

Figure 3 PKA APIs overview



CAUTION:

Provide support for True Random Number Generation (TRNG) operations.

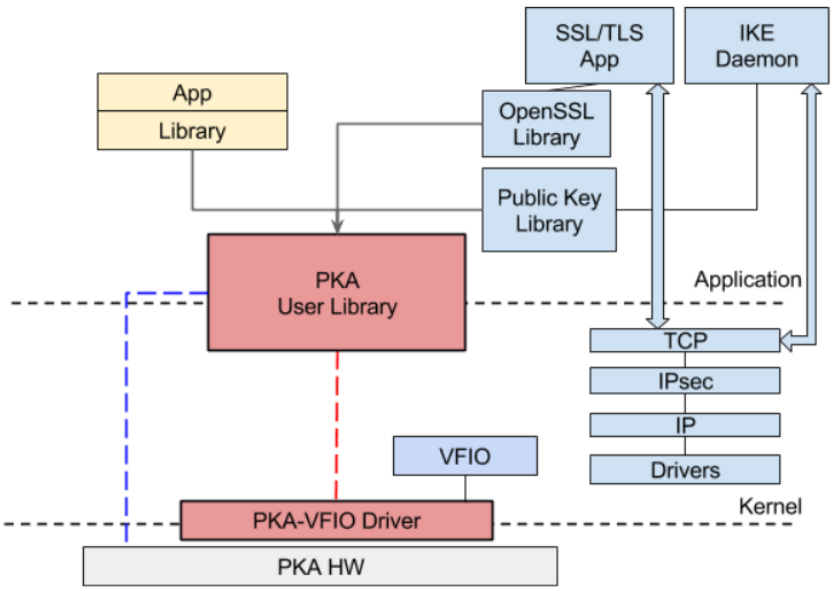
2.2 PKA Implementation

NOTE:

The current PKA implementation does not yet provide complete support for either OpenSSL or kernel Crypto API. It defines user interfaces only that would be used over user space application requiring PK acceleration.

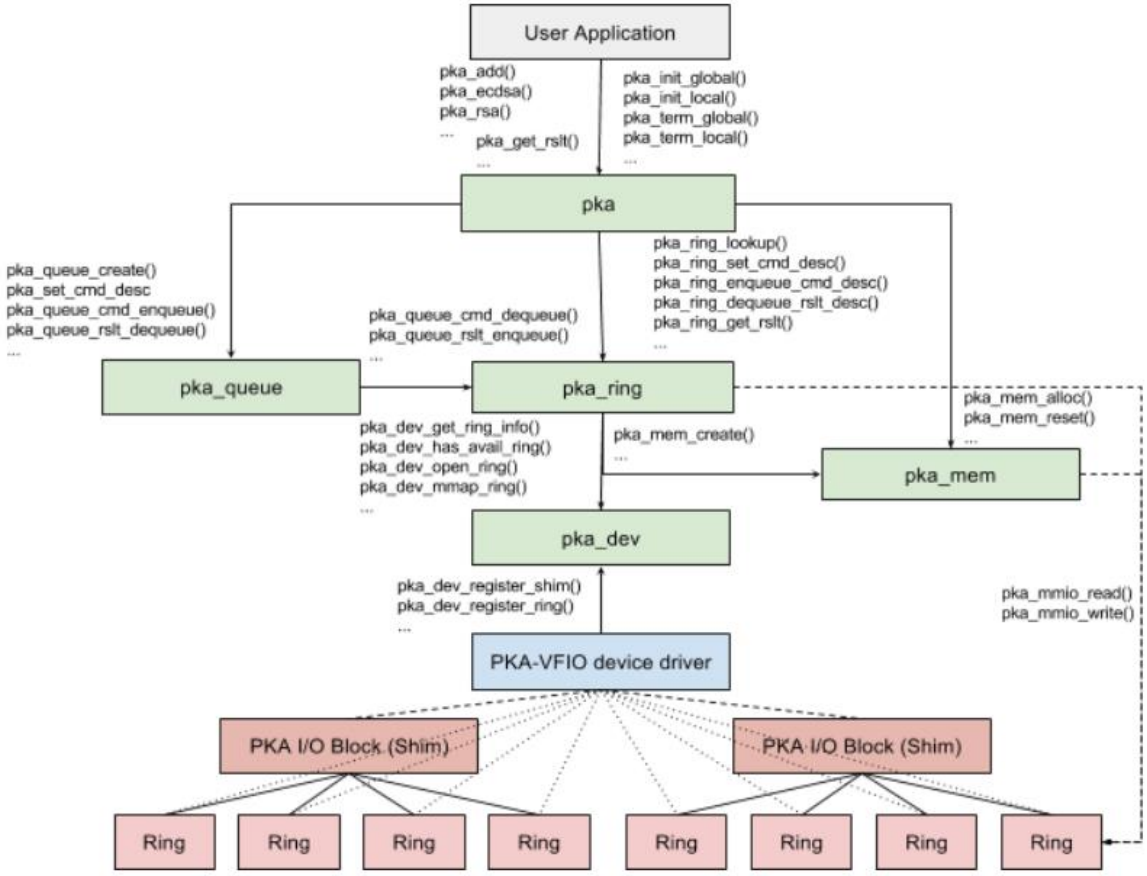
The PKA framework might be considered as a modular architecture among both kernel and user spaces. Kernel component and APIs are implemented multiple features that enable PK hardware acceleration. This is realized through internal components, i.e., a set of libraries that provide all the elements needed for handling PK commands, dealing with multithreading, and managing hardware resources. It also includes a VFIO-based device driver to ensure the development of safe, high performance user-space drivers. The driver implements an interface that might be supported by virtualization tools and some generic driver infrastructure. It breaks the hardware resources into a set of devices which are isolatable from all other devices in the system, and that might be allocated to different applications.

Figure 4 PKA Driver Architecture



The PKA user library provides interfaces that allow applications to initialize an execution context with a single or multiple Rings, where Rings might belong to different I/O blocks. The library allocates then Ring resources to the given application and processes different PK operations. To illustrate that, we present different PKA calls through the proposed interfaces.

Figure 5 PKA user library calls



2.2.1 Kernel component

BlueField has two crypto blocks, each consists of two PKA blocks based on EIP-154. The EIP-154 uses four rings as a communication mechanism between its farm engines controlled by the master firmware and the ARM cores.

PKA devices are managed by the **'pka-mlxbf'** Linux driver. The driver is designed to handle code for different type of devices; The PKA block is considered as the main device that might be shared among several PK user instances. In order to realize this vision, a character device file is created for each ring within a PKA block. That ends up with 16 devices that might be assigned separately regardless the user instance requesting them. However, the main device must be initialized before running any instance or using any ring. Thus, the driver deals with those two kind of devices, the main PKA block referred to as shim and the ring devices. Note that the ring is also referred to as VFIO device.

2.2.1.1 Device probe

The Linux device driver provides two probe routines: ***pka_drv_probe_device()*** and ***pka_drv_probe_vfio_device()***. The first routine probes and registers the given shim, and the second routine initializes and registers the given ring. During shim registration, the boot-up sequence is executed, i.e., loading firmware images into internal rams, configuration of shim's parameters and mapping of memory regions and control/status registers. The shim is registered through the ***pka_dev_register_shim()*** call. On the other hand, ring devices are registered as

VFIO devices. Each device belongs to a VFIO group that is the unit for device allocation. Note that it is mandatory that device is associated with an IOMMU group before adding it to a VFIO group. The IOMMU group for a given device is retrieved through ***vfio_iommu_group_get()***, and then, the device is added to a VFIO group through ***vfio_add_group_dev()***. The ring registration is done by calling ***pka_dev_register_ring()***; Ring counters are reset, the buffer memory words are configured with regards to the defined configuration. You can check the configuration file for further details about the shim and ring configuration; the file is located in *src/include/pka_config.h*.

2.2.1.2 Driver commands

Moreover, the driver implements the VFIO device operations including ***open(), release(), mmap()*** and ***ioctl()***. Among others, the ***ioctl()*** call supports PK driver specific commands other than the commands defined for VFIO groups and devices. Those commands are ***PKA_VFIO_GET_REGION_INFO***, to allow user space to get information about device memory, in particular the ring counters region and the window RAM region, and ***PKA_VFIO_GET_RING_INFO,*** to allow user space to get ring configuration from the EIP-154 Buffer RAM.

2.2.1.3 Interrupt handling

All ring devices are accessed by polling. However, the driver registers an IRQ handler for shim devices. Those interrupts are not really relevant to the library or the user application, so the handler will simply disable the source of interrupt and will quickly return.

2.2.1.4 ACPI support

The driver also implements a DT-based probe routine and an ACPI-based probe routine. Since there can be two kernel build combinations, it is possible to probe devices in both cases.

NOTE:

Currently, the DT type routine is not implemented yet, due to a couple of issues related to the SMMU configuration and the kernel source version.

2.2.2 User Library

The API enables PK hardware acceleration through internal components, i.e., a set of libraries that provide all the elements needed for handling PK commands, dealing with multithreading, and managing hardware resources.

2.2.2.1 User API

It presents mainly an interface to receive PK requests, and to format these requests according to command descriptors. A generic layout of a PKI command descriptor can be found in (Firmware Reference Manual, 2013).

This API makes available a number of arithmetic (both basic operations (e.g., add and multipy) as well as complex operations (e.g., modular exponentiation and modular inversion). It is used for high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptology, and the Federal Digital Signature Algorithm (DSA as documented in FIPS 186) public-private key systems.

It consists of a northbound interface which allows user application to submit PK operations. Note that almost all functions are asynchronous and require a 'handle' argument to be passed to them. Typically, a 'handle' structure encapsulates all the parameters, status of a PK instance owned by a given user application. Note that application running multiple threads has many handles as threads.

First of all, *pka_init_global()* gets called to initialize the crypto instance. User application uses a PKA instance to request PK operations. An instance refer to a global execution context including a set of hardware rings and data structures specific to the application. The *pka_init_global()* call then creates a shared memory object which might be accessed by both processes and threads associated with the same PK instance. The library determines the size of shared memory needed by the instance according to the size and the number of SW queues. Indeed, user application must specify SW queues parameters that are assigned to its worker threads. Each worker thread is characterized by its command queue and result queue. *pka_init_global()* also lookups for available HW Rings, creates a set of SW queues and returns a pointer to the shared memory referred by *pka_global_info_t*.

Below the definition of the PK global information structure encapsulated by the *pka_instance_t*:

```
typedef struct
{
    pid_t            main_pid;          ///< main process identifier

    uint32_t         requests_cnt;      ///< command request counter.
    uint32_t         queues_cnt;        ///< number of queues supported.
    uint32_t         cmd_queue_size;    ///< size of a command queue.
    uint32_t         rslt_queue_size;   ///< size of a result queue.

    pka_atomic32_t   workers_cnt;       ///< number of active workers.
    pka_worker_t     workers[PKA_MAX_QUEUES_NUM]; ///< table of initialized
                                                  ///  thread workers.

    uint32_t         rings_byte_order;  ///< byte order whether BE or LE.
    uint32_t         rings_mask;        ///< bitmask of allocated HW rings.
    uint32_t         rings_cnt;         ///< number of allocated Rings.
    pka_ring_info_t  rings[PKA_MAX_NUM_RINGS];    ///< table of allocated rings
                                                  ///  to process PK commands.

    pka_shmem_info_t shmem_info;        ///< shared memory information.

    /// Lock-free implementations have higher performance and scale better
    /// than implementations using locks. User can decide whether to use
    /// lock-free implementation or its own locking mechanism by setting flags.
    /// these flags tend to optimize performance on platforms that implement
    /// a performance critical operation using locks.
    pka_atomic64_t   lock;              ///< protect shared resources.
    uint8_t          flags;             ///< flags supplied during creation.

    uint8_t         *mem_ptr;           ///< pointer to free memory space of
                                        ///  SW queues.

    uint8_t mem[0] __pka_cache_aligned; ///< memory space of SW queues starts
                                        ///  here.
} pka_global_info_t;
```

After calling *pka_init_global()*, a client can access Ring resources through a 'handle'. To obtain this handle, one might call *pka_init_local()* which requires a valid PK instance. The handle encapsulates local execution context information, provide a reference to PK global information, and allows to process PK operation. From user point of view, PK operation consists on filling command descriptor(s) according to the given PK command and retrieving the associated results. Descriptors contains only pointers to vector data in window RAM. When a result is ready, the API allows client to read result descriptor and vectors data.

Below the definition of the PK local information structure encapsulated by the handle *pka_handle_t* :

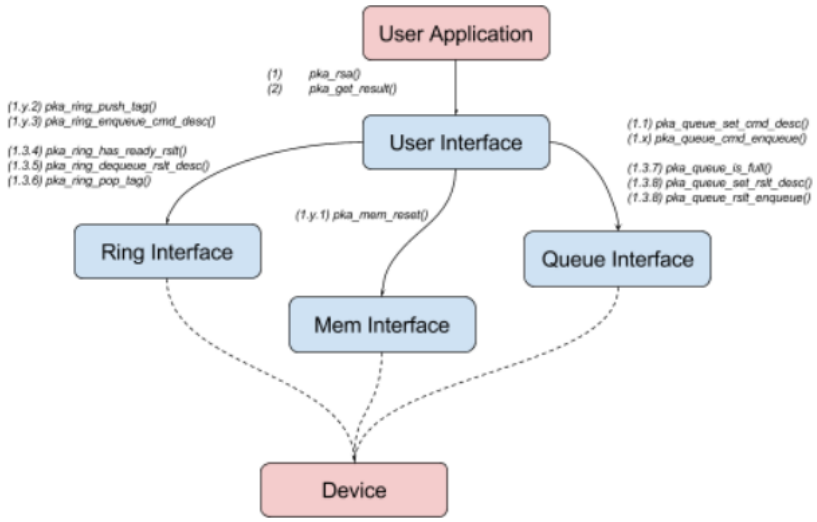
```
typedef struct
{
    uint32_t            id;         ///< handle identifier - thread specific.
    uint32_t            req_num;    ///< number of outstanding requests.
    pka_global_info_t  *gbl_info;   ///< pointer to the instance information the
                                    ///  handle belongs to.
} pka_local_info_t;
```

All functions used to perform PK commands are asynchronous and require a 'handle' argument to be passed to them. In addition to the 'handle', the function takes a pointer to user application data that might be associated with the PK operation, e.g., crypto session information.

Upon a PK command issued, The API processes the operands byte order, if needed. It also, apply few operand checks in order to prevent errors in hardware. Note that all of the operands passed via the PKA API calls are copied from the user context to the PKA instance context. After that, the command is submitted via ***pka_submit_cmd()*** call. This function retrieves both of the local and global information related to the handle and instance, respectively. It presents the main function for PK command processing.

First, the function prepares the command descriptor to enqueue via ***pka_queue_set_cmd_desc().*** This descriptor encapsulates all of the operands and meta-data needed for command processing. When the synchronization mode is disabled then simply add the command descriptor to the ring, if there are available descriptors. Otherwise, add it to the worker queue. After that, the current worker is responsible of retrieving results from the result rings and appending them to their associated workers' reply queue. This is done by the function ***pka_rslt_dequeue()***. After that, the current worker processes all of the command queues that belong to its instance using ***pka_process_queues_nosync()***.

Note that the queue descriptors and the ring descriptors have different format. Unlike the ring descriptors which are hardware specific, queue descriptors holds user-related data and are designed to minimize the overhead introduced by enqueuing/dequeuing objects. Ring descriptor defines a 64-bit field referred to as tag, which might be used by user applications. Therefore, the PKA library use that field to store a pointer to data related to the worker and few command statistics. When dequeuing objects from the ring, the worker result queue can be easily retrieved via that data pointer. A generic layout of a ring descriptors can be found in (Firmware Reference Manual, 2013). Once all of the result rings processed, the worker is then free and can perform either other non-PKA related tasks or additional PKA related tasks.



On the other hand, when the synchronization mode is enabled, the current worker tries to acquire a lock via ***pka_try_acquire_lock().*** This lock is provided by the instance to protect accesses to shared rings. Technically, it holds information about the worker requesting the lock and the lock status, whether acquired or not. When a given worker tries to acquire the lock it atomically sets the bottom byte to its number (identifier + 1, to allow the possibility that worker's number starts at 0). But this will only succeed if this bottom byte is zero. If the lock is already held by another worker (i.e., bottom byte is non-zero). The remaining lock bit field are set with the dedicated thread request bit. Those bits are set so that the current lock owner will know about other requests, in other words, the lock owner will not be able to release this lock while any of these request bits are set. Releasing the lock is done by calling ***pka_try_release_lock().*** Acquiring and/or releasing the lock is based on an Exclusive Load/Store instructions. Such locking mechanism leverages the cases where there may still have work to do, before releasing the lock. For instance, when the worker 'x' owns the lock, this latter is responsible of enqueuing/dequeuing objects to/from rings. If another worker 'y' tries to acquire a lock but fails to do that, it will append its request to the command queue, and if this was done few cycles before the worker 'x' tries to release the lock, then, the request will wait longer until another worker 'z' acquires the lock and process the command queues. Thanks to the request bit field within the lock, that scenario is avoided and all of the requests are processed by the lock owner before exiting.

Note that those who failed on their first attempt to acquire the lock, will make a second attempt, to register their request, in case they fail a second time. This second call waits, if necessary, for any previous request bit to be cleared. Each bit setting corresponds to exactly one PK request. It is somewhat rare that the second attempt to acquire the lock fails. Indeed, the current lock owner is guaranteed to see the previously set request bit and act on it before it can release the lock.

The synchronization mode aims then, to avoid multiple accesses to rings and optimize the queue and ring processing among workers. If disabled, the application should provide its own mechanism to prevent two workers of accessing rings at the same time.

Command queue processing is done by calling ***pka_process_queues_sync().*** This function will make sure that the current lock owner has no pending request before releasing the lock. Note that the queue processing scheme is practically the same either the synchronization mode is enabled or not; the ring result queues are processed first in order to get available results, the results are appended to the reply queue. Then, the command queues are processed while there

are available rooms in the command rings. To check the number of outstanding command requests for a given 'handle', one might call ***pka_request_count()*.**

2.2.2.2 Queue Management API

When client applications run over multiple threads, each thread can use the attached instance to request PK operations (still applicable when client runs a single thread). Thus, PKA library assigns queues for threads and provides functions to handle these queues. Each queue is associated to a single thread and define a maximum number of elements (or threshold). For each input queue there is an associated output queue. This helps to isolate thread operations but requires more resources. Note that there is no defined priority per client/thread.

This API consists of an implementation of circular queues on top of command/result descriptor rings. It allows multiple threads/workers to submit PK commands to the hardware without causing ring congestion. Software-based queues are assigned to clients which may run over single or multiple threads, a pair of queue per thread: one queue to append command descriptors and another one to append result descriptors. Each group of queues is associated to one or a group of rings depending on client execution context a.k.a PK instance. The implementation of the software-based queues help to leverage the small size of descriptor rings and avoid interrupts, so far (processes have to wait until a given ring can accept new descriptors again). Queues have the following properties :

- FIFO,
- Capacity is fixed,
- Lockless implementation,

However, having many circular queues with significant size costs in terms of memory (more than linked list queue). An empty queue contains at least N objects.

Also note that the implementation may include a mechanism which exert a back pressure to inform a given client to pause. It defines a threshold, once an enqueue reaches the high threshold, the client is notified.

```
typedef struct
{
    uint32_t cnt;              ///< Number of items in the queue.
    uint8_t  flags;            ///< Flags supplied at creation.

    // Queue producer status.
    struct prod
    {
        uint32_t watermark;        ///< Maximum items before EDQUOT.
        uint32_t sp_enqueue;       ///< True, if single producer.
        uint32_t size;             ///< Size of the queue.
        uint32_t mask;             ///< Mask (size-1) of queue.
        volatile uint32_t head;    ///< Producer head.
        volatile uint32_t tail;    ///< Producer tail.
    } prod __pka_cache_aligned;

    // Queue consumer status.
    struct cons
    {
        uint32_t sc_dequeue;       ///< True, if single consumer.
        uint32_t size;             ///< Size of the queue.
        uint32_t mask;             ///< Mask (size-1) of queue.
        volatile uint32_t head;    ///< Consumer head.
        volatile uint32_t tail;    ///< Consumer tail.
    } cons;

#ifdef PKA_LIB_QUEUE_DEBUG
    struct pka_queue_debug_stats stats;
#endif

    uint8_t queue[0] __pka_cache_aligned;///< Memory space of queue starts here.
                                         /// not volatile so need to be careful
                                         /// about compiler re-ordering.
} pka_queue_t;
```

The current implementation supports simple producer and simple consumer. One object can be enqueued and dequeued at a time.

2.2.2.3 Ring management API

The window RAM supports 4 Rings, it is statically divided into 4 separate (contiguous or non-contiguous) regions of 16KB. Each region include command/descriptor rings and operands. Thus, the driver partitions the 16KB memory region with descriptors memory and vector data memory. Descriptors memory supports the 4 Rings and usually occupies a small region. While the library is responsible for selecting the location of command/result descriptor rings within the allocated memory (This is done by writing Buffer RAM words), it is mandatory to specify if each pair of descriptor rings are co-located or separated. Co-located rings offer a better memory usage by saving ½ of descriptor memory space. On the other hand, it causes performance issues where all commands hold by the rings should be processed before copying result descriptors. Separated descriptor rings provide a fast response, however it increases memory usage, i.e., descriptor memory requires more space and thus operand memory should decrease. In addition, the library allocates and frees memory for input/output data vectors which consist of operands and results.

NOTE:

Co-located or separated descriptor rings should be considered. Currently, no decision was made regarding this point.

Moreover, a concrete strategy for memory partition should be provided, explained and justified. We have to determine the maximal size of operands for a each operation, approximate the number of occurrence of the given operation and evaluate the number of descriptors/commands which could be processed. The initial proposition is to partition with 1K-2K for descriptor memory and with the remaining 15-14K for vector data memory

An efficient memory partition is a trade-off between descriptors a given Ring can hold, and vectors corresponding to the descriptors a vector data memory can support. Thus, it is possible to have the rings full while the vector memory can support more data. The opposite can also happen, but it is not suitable.

We statically divide the 16K memory region into three partitions: First partition is reserved for command descriptor ring (1K), second partition is reserved for result descriptor ring (1K), and the remaining 14K are reserved for vector data. Through this memory partition scheme, command/result descriptor rings hold a total of 1KB/64B = 16 descriptors each. The addresses for the rings start at offset 0x3800. Also note that it is possible to have rings full while the vector data can support more data, the opposite can also happen, but it is not suitable. For instance ECC point multiplication requires 8 input vectors and 2 output vectors, a total of 10 vectors. If each vector has a length of 24 words (17x4B = 68B), we can process 14KB/680B = 21 operations which is close to 16 the total descriptors supported by rings. On the other hand, using 12K vector data region, allows to process only 18 operations, while rings can hold 32 descriptors (ring usage is significantly low). For ECDSA verify, we have 12 vectors which require 816B, with 14KB we can handle 17 operations, against 15 operations with 12KB vector data memory. We believe that the aforementioned memory partition help us to leverage the trade-off between supported descriptors and required vectors. Note that these examples gives an approximate values and does not include buffer word padding across vectors.

To request rings, a user application must call ***pka_ring_lookup().*** This call returns a table of rings matching the number specified by the initial request or less than that, when the number of available rings are less than the requested number. Based on the requested number of rings, the function looks over the rings, issues library calls to the device interface to determine whether a given ring can be used or not. A call to ***pka_dev_has_avail_ring()*** will trigger a modulus-based algorithm (RR like) to rolls over the PKA crypto blocks, and tries to open one ring at a time. If it succeeds, then the ring is added to the returned list of rings. Otherwise, the algorithm continues and check the following ring within the next block. To give an overview about the algorithm, the current configuration defines N=4 PKA crypto blocks, each block has M=4 rings; if the application requests X=6 rings, then the algorithm will return the following rings: Rings (N, M) = {(0, 0), (1, 0), (2, 0), (3, 0), (0, 1), (1, 1)}, in case all of the rings are available.

This API also provides a support for descriptors management within Rings. Upon a descriptor is ready, it should be written to command descriptor ring. A command descriptor ring holds several descriptors. This API defines functions to append command descriptors to rings, read result descriptors and write to/from window RAM.

```
/// This structure declares ring parameters which can be used by user interface.
typedef struct
{
    int         fd;             ///< file descriptor.
    int         group;          ///< iommu group.
    int         container;      ///< vfio cointainer

    uint32_t    idx;            ///< ring index.
    uint32_t    ring_id;        ///< hardware ring identifier.

    uint64_t    mem_off;        ///< offset specific to window RAM region.
    uint64_t    mem_addr;       ///< window RAM region address.
    uint64_t    mem_size;       ///< window RAM region size.

    uint64_t    reg_off;        ///< offset specific to count registers region.
    uint64_t    reg_addr;       ///< count registers region address.
    uint64_t    reg_size;       ///< count registers region size.

    void       *mem_ptr;        ///< pointer to map-ped memory region.
    void       *reg_ptr;        ///< pointer to map-ped counters region.

    pka_ring_desc_t ring_desc;  ///< ring descriptor.

#ifdef PKA_LIB_RING_DEBUG
    struct pka_ring_debug_stats stats;
#endif

    uint8_t     big_endian;     ///< big endian byte order when enabled.

} pka_ring_info_t;
```

Note that functions and structures used by this API are already implemented for userland applications. However, code can be arranged and somehow extended to be called within kernel module code.

2.2.2.4 Memory management API

The memory management interface is used by rings to allocate free memory needed by PK commands. PKA memory allocator's job is primarily to manage the data memory - i.e. efficiently allocate and free memory space to hold the input/output vectors. One could use this code to do individual allocations and frees for each vector, but instead it is expected that a single contiguous allocation/free will be done for all the vectors - i.e. operands and results, belonging to a single command. It is possible to also support a mode of operation, whereby individual operand allocation can be used when a single command allocation fails for lack of memory (i.e. this can deal efficiently with the occasional data memory fragmentation where there is enough contiguous memory pieces to hold the individual operand, but not single piece large enough to hold all of the operands).

This API assumes that Data Memory is in the bottom 14KB of the "PKA window RAM" and so the addresses for the rings start at offset 0x3800. Also, note that just because the rings hold 16 descriptors, does not mean that 16 commands can be outstanding - since it is expected that often the Data Memory will run out before any or all of the rings are full themselves. Of course the opposite can also happen (though less likely) - that is the rings are full, when the Data Memory is not!

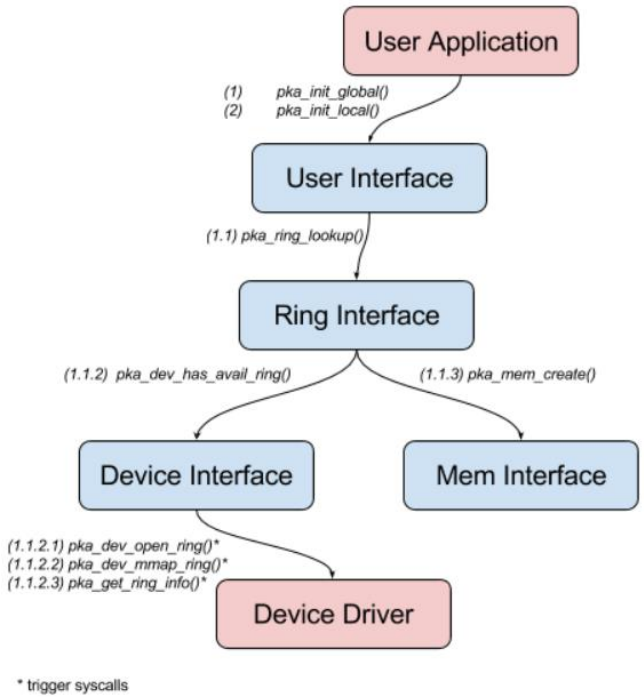
All allocations handled by the current implementation start at least on 64-byte boundaries and all allocation have sizes that are a multiple of 64 bytes. The algorithm here always maximally coalesces contiguous free space. In other words, there is never a case where two free space descriptors point to adjacent memory. Of course the converse is not true. Used space blocks can be adjacent to either other used space blocks to free space blocks.

2.2.2.5 Device Management API

The implementation of the device driver is based on the Linux VFIO. It is essentially an interface that can be implemented by device drivers and that is supported by virtualization tools and some generic driver infrastructure. VFIO defines a set of devices which are isolatable from all other devices in the system as a "group", and a container class to hold one or more groups. A container is created by simply opening the */dev/vfio/vfio* character device. Once a new group is added to the container and the related device is bound to the VFIO driver, it will appear as */dev/vfio/group*, where *group* is the IOMMU group number of which the device is a member. More details can be found in VFIO kernel Documentation[3].

To meet the driver model principles, we define a device interface which hold data structure and functions used by device driver to register/unregister devices. The PKA device interface consists of a southbound interface of the PK library for requesting and managing hardware resources. This also define the code to initialize and prepare devices to be used by crypto instances. After device initialization, Ring resources can be opened and Ring regions can be mapped into physical address space, so clients can read/write from/into device registers/memory directly.

It implements ***pka_dev_open_ring()*** to open a given ring from the userland. This call makes several syscalls to open the IOMMU group associated with the ring, test if the group is viable and available, add the group to the container, set an IOMMU type and returns a corresponding file descriptor for the device. Before calling ***pka_dev_open_ring()*** the PKA library passes an already opened container. A PK instance opens a single container for its set of ring devices. Ring regions are mapped via ***pka_dev_mmap_ring(),*** this call insure the mapping of control/status registers region and window RAM region only. User application are not able to access ring information control/status words located in Buffer RAM.



[3] https://www.kernel.org/doc/Documentation/vfio.txt

Evaluation Model

(Evaluation model --description goes here)

2.3 Benchmark

Public Key operations:

- RSA
- DSA sign/verify,
- ECDSA sign/verify

Performance metric:

- Number of operation per second (straightforward metric)

Controlled parameters:

- key size (exponent) (1024 bit / 2048 bit / 8 bit)
- key size (modulus) (1024 bit keys are common default / 2048 bit)

(typically public key size and private key size are the same in p2p)

Observed parameter:

- CPU overhead : Time to load operands, Time to set descriptors, Waiting time in SW queue

(This parameter should be considered in order to enhance/optimize hardware acceleration)

2.4 Results

Below, we measure the overhead due to the HW offload, running one thread/one Ring in single process mode, on a 2 GHz Cortex-A72.

Previous implementation:

PK command	Size (bits)	Sign (cycles)	Verify (cycles)
RSA	***1024***	3120	
DSA	***1024***	1680	3280
	2048	2100	2640
	3072	2480	3140
ECDSA	***256***	1860	3460

	384	3480	3760
	521	2580	2940

Recent implementation:

PK command	Size (bits)	Sign (cycles)	Verify (cycles)
RSA	***1024***	3700	
DSA	***1024***	1660	3740
	2048	3820	4040
	3072	2340	2600
ECDSA	***256***	1600	3780
	384	2040	2520
	521	2420	3060

3 References

Firmware Reference Manual. (2013, 12 09). *Public Key Infrastructure Engine*. INSIDE Secure.

Lumpkin, T., & Phillips, K. (2006). Linux, IPsec, and Crypto Hardware Acceleration. Citeseer.

OpenSSL ENGINE. (n.d.). Retrieved from https://www.openssl.org/docs/man1.0.1/crypto/engine.html