|  |
| --- |
| Diamond Key Security  Copyright © 2018, 2019 Diamond Key Security, NFP |
| CrypTech Alpha Software Developer Reference Guide |



Change Log

|  |  |
| --- | --- |
| Date | Description |
| 7/2/2018 | Initial document. |
|  |  |
|  |  |
|  |  |

Table of Contents

[Change Log 2](#_Toc518316683)

[1. Introduction 5](#_Toc518316684)

[1.1. Scope of Document 5](#_Toc518316685)

[1.2. Terms and Definitions 5](#_Toc518316686)

[1.3. CrypTech Alpha Hardware Description and Relationships 5](#_Toc518316687)

[2. CrypTech Repositories 6](#_Toc518316688)

[2.1. libhal 6](#_Toc518316689)

[2.1.1. Remote Procedure Calls – RPC API 6](#_Toc518316690)

[2.1.2. Repository README from CrypTech 13](#_Toc518316691)

[2.2. stm32 16](#_Toc518316692)

[2.2.1. Firmware Details 16](#_Toc518316693)

[2.2.2. Repository README from CrypTech 17](#_Toc518316694)

[2.3. pkcs11 20](#_Toc518316695)

[2.3.1. Implemented PKCS#11 Functions 20](#_Toc518316696)

[2.3.2. Unimplemented PKCS#11 Functions 24](#_Toc518316697)

[2.3.3. Repository README from CrypTech 27](#_Toc518316698)

[2.4. tamper 29](#_Toc518316699)

[2.4.1. Repository README from CrypTech 29](#_Toc518316700)

[3. Key CrypTech Software 31](#_Toc518316701)

[3.1. ‘cryptech\_probe’ (Python) 31](#_Toc518316702)

[3.2. ‘cryptech\_muxd’ (Python) 32](#_Toc518316703)

[3.2.1. SerialIOStream (tornado.iostream.BaseIOStream) 32](#_Toc518316704)

[3.2.2. PFUnixServer (tornado.tcpserver.TCPServer) 32](#_Toc518316705)

[3.2.3. RPCIOStream (SerialIOStream) 32](#_Toc518316706)

[3.2.4. RPCServer (PFUnixServer) 32](#_Toc518316707)

[3.2.5. CTYIOStream (SerialIOStream) 32](#_Toc518316708)

[3.2.6. CTYServer (PFUnixServer) 33](#_Toc518316709)

[3.2.7. Start up algorithm 33](#_Toc518316710)

[3.3. ‘cryptech\_console’ (Python) 33](#_Toc518316711)

[3.4. ‘cryptech\_upload’ (Python) 33](#_Toc518316712)

[3.5. ‘libhal.py’ (Python) 34](#_Toc518316713)

[3.6. ‘cryptech\_backup’ (Python) 34](#_Toc518316714)

[3.7. libcryptech-pkcs11.so (C) 35](#_Toc518316715)

[A. CrypTech Software Summary 36](#_Toc518316716)

[B. Alpha Board Drawing 37](#_Toc518316717)

[C. SLIP 38](#_Toc518316718)

[D. PKCS 11 Functions 39](#_Toc518316719)

[E. Tornado 42](#_Toc518316720)

[F. External Data Representation (XDR) 43](#_Toc518316721)

[G. References 43](#_Toc518316722)

1. Introduction

Diamond Key Security (DKS) is bringing to market a unique set of security products that are designed and built upon the open and transparent CrypTech technology for a multitude of enterprise and individual applications. DKS is utilizing open source software from CrypTech and likely other sources to create revolutionary security solutions. The CrypTech project initiative has succeeded in building a generalized module for cryptography functions that underpin secure communications and Internet technologies.

* 1. Scope of Document

This is a programmer’s reference manual designed to explain the CrypTech software the runs on the host computer including the PKCS#11 shared library. This is not a API reference even though it gives some API details. It goes beneath the surface to explain how the code works and how different parts are related to each other. This reference is intended to serve as a map of the key areas of the code base so that new developers will be able to know how to extend the code for new features and how to analyze the code for fixes and enhancements. The following Cryptech repositories will be considered:

* sw/libhal
* sw/pkcs11
* sw/stm32
  1. Terms and Definitions

AF\_UNIX – AF\_UNIX creates file system objects and only works between processes on the same host.

CTY – Console Port on HSM – This is the management port that behaves like a console the reads text based commands and returns information as plain text.

FPGA – Field Programmable Gate Array

PySerial – Python serial port access library – This module encapsulates the access for the serial port. It provides for Python running on Windows, OSX, and BSD. iii

RPC – RPC Port on the HSM – This is the user port and it processes data as RPC request

SLIP – Serial Line Internet Protocol

Tornado – Tornado is a Python web framework and asynchronous networking library.

* 1. CrypTech Alpha Hardware Description and Relationships

To understand the source code folder structure and the purpose of much projects a basic understanding of the hardware is needed. The CrypTech Alpha is a custom HSM board with three main components: the FPGA sub system, an ARM processor, and a tamper circuit. i

**The FPGA Sub System – Xilinx XC7A200T**

FPGA stands for Field Programmable Gate Array and the alpha board uses one to implement CrypTech crypto/security cores accessible by the CPU as coprocessors.i This is programmed using Verilog.

**ARM Processor – STM32F42**

The ARM processor talks to host systems and handles incoming commands. Basically implements the application interface. Controls the FPGA Sub System. The 2 USB ports connect to the board and talk to the STM meaning the STM interface in the source code must handle all input. i The firmware runs on the STM.

**Tamper Circuit – ATtiny828**

The tamper circuit is resposible for implementing tamper detection and control/alarm as a separate functionality from the CPU. On the Alpha board this system is fairly simplistic. But we want to at least have a minor MCU that can run idependently on batter power and control the Master Key Memory (MKM), detect external events and generate alarms. This allows us to start developing and reason about tamper detection and monitoring separatel0y from the CPU. i

1. CrypTech Repositories
   1. libhal

‘libhal’ creates a library that can be statically linked into a program. It is used by the STM32 firmware as well as by the PKCS#11 shared library. It contains low-level code used to perform the cryptographics functions with the CrypTech FPGA. It also has the higher-level remote procedure call (RPC)

<https://git.cryptech.is/sw/libhal.git>

* + 1. Remote Procedure Calls – RPC API

A remote procedure call is when a computer program causes a procedure to execute in a different address space. The RPC API provides the interface that both the firmware and external programs such as the PKCS#11 shared library use.

* + - 1. RPC Function List ‘sw/libhal/hal.h’

The interface is defined in ‘sw/libhal/hal.h’ and included the following functions:

extern hal\_error\_t hal\_rpc\_set\_pin(const hal\_client\_handle\_t client,

const hal\_user\_t user,

const char \* const newpin, const size\_t newpin\_len);

Sets the PIN for a user. The client parameter is an extension to PKCS#11 intended to support multiple PKCS#11 using applications sharing an HSM.

extern hal\_error\_t hal\_rpc\_login(const hal\_client\_handle\_t client,

const hal\_user\_t user,

const char \* const pin, const size\_t pin\_len);

Logs into the HSM using a user and PIN combination for a client application. The client parameter is an extension to PKCS#11 intended to support multiple PKCS#11 using applications sharing an HSM.

extern hal\_error\_t hal\_rpc\_logout(const hal\_client\_handle\_t client);

Logs out from the HSM for a client application. The client parameter is an extension to PKCS#11 intended to support multiple PKCS#11 using applications sharing an HSM.

extern hal\_error\_t hal\_rpc\_logout\_all(void);

Logs out all clients. The Cryptech alpha firmware uses clients as an extension to PKCS#11 to support multiple PKCS#11 using applications to share an HSM.

extern hal\_error\_t hal\_rpc\_is\_logged\_in(const hal\_client\_handle\_t client,

const hal\_user\_t user);

Returns HAL\_OK if the user has login in for a client. The client parameter is an extension to PKCS#11 intended to support multiple PKCS#11 using applications sharing an HSM

extern hal\_error\_t hal\_rpc\_get\_version(uint32\_t \*version);

Returns the RPC version used by the firmware.

extern hal\_error\_t hal\_rpc\_get\_random(void \*buffer, const size\_t length);

Fills a buffer with random data.

extern hal\_error\_t hal\_rpc\_hash\_get\_digest\_length(const hal\_digest\_algorithm\_t alg, size\_t \*length);

Returns the length of the digest for a hash algorithm.

extern hal\_error\_t hal\_rpc\_hash\_get\_digest\_algorithm\_id(const hal\_digest\_algorithm\_t alg,

uint8\_t \*id, size\_t \*len, const size\_t len\_max)

Copies the hash digest for the given algorithm into the buffer specified by id.

extern hal\_error\_t hal\_rpc\_hash\_get\_algorithm(const hal\_hash\_handle\_t hash, hal\_digest\_algorithm\_t \*alg)

Returns the algorithm by the hash handle of the operation.

extern hal\_error\_t hal\_rpc\_hash\_initialize(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_hash\_handle\_t \*hash,

const hal\_digest\_algorithm\_t alg,

const uint8\_t \* const key, const size\_t key\_length);

Initializes a multi-part hash or hmac operation using the given parameters and returns a handle.

extern hal\_error\_t hal\_rpc\_hash\_update(const hal\_hash\_handle\_t hash,

const uint8\_t \* data, const size\_t length);

Updates the hash operation with data to be run the hashing algorithm on.

extern hal\_error\_t hal\_rpc\_hash\_finalize(const hal\_hash\_handle\_t hash,

uint8\_t \*digest, const size\_t length);

Finalizes a multi-part hashing procedure and returns the digest.

extern hal\_error\_t hal\_rpc\_pkey\_load(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_pkey\_handle\_t \*pkey,

hal\_uuid\_t \*name,

const uint8\_t \* const der, const size\_t der\_len,

const hal\_key\_flags\_t flags);

Receive key from application, generate a name (UUID), store it, and return a key handle and the name.

extern hal\_error\_t hal\_rpc\_pkey\_open(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_pkey\_handle\_t \*pkey,

const hal\_uuid\_t \* const name);

Look up a key given its name, return a key handle.

extern hal\_error\_t hal\_rpc\_pkey\_generate\_rsa(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_pkey\_handle\_t \*pkey,

hal\_uuid\_t \*name,

const unsigned key\_length,

const uint8\_t \* const public\_exponent,

const size\_t public\_exponent\_len,

const hal\_key\_flags\_t flags);

Generate a new RSA key with supplied name, return a key handle.

extern hal\_error\_t hal\_rpc\_pkey\_generate\_ec(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_pkey\_handle\_t \*pkey,

hal\_uuid\_t \*name,

const hal\_curve\_name\_t curve,

const hal\_key\_flags\_t flags);

Generate a new EC key with supplied name, return a key handle. At the moment, EC key == ECDSA key, but this is subject to change.

extern hal\_error\_t hal\_rpc\_pkey\_close(const hal\_pkey\_handle\_t pkey);

Discard key handle, leaving key intact.

extern hal\_error\_t hal\_rpc\_pkey\_delete(const hal\_pkey\_handle\_t pkey);

Delete a key from the store, given its key handle.

extern hal\_error\_t hal\_rpc\_pkey\_get\_key\_type(const hal\_pkey\_handle\_t pkey,

hal\_key\_type\_t \*type);

Get type of key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_get\_key\_curve(const hal\_pkey\_handle\_t pkey,

hal\_curve\_name\_t \*curve);

Get curve of key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_get\_key\_flags(const hal\_pkey\_handle\_t pkey,

hal\_key\_flags\_t \*flags);

Get flags of key associated with handle.

extern size\_t hal\_rpc\_pkey\_get\_public\_key\_len(const hal\_pkey\_handle\_t pkey);

Get length of public key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_get\_public\_key(const hal\_pkey\_handle\_t pkey,

uint8\_t \*der, size\_t \*der\_len, const size\_t der\_max);

Get public key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_sign(const hal\_pkey\_handle\_t pkey,

const hal\_hash\_handle\_t hash,

const uint8\_t \* const input, const size\_t input\_len,

uint8\_t \* signature, size\_t \*signature\_len,

const size\_t signature\_max);

Sign something using private key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_verify(const hal\_pkey\_handle\_t pkey,

const hal\_hash\_handle\_t hash,

const uint8\_t \* const input, const size\_t input\_len,

const uint8\_t \* const signature, const size\_t signature\_len);

Verify something using public key associated with handle.

extern hal\_error\_t hal\_rpc\_pkey\_match(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

const hal\_key\_type\_t type,

const hal\_curve\_name\_t curve,

const hal\_key\_flags\_t mask,

const hal\_key\_flags\_t flags,

const hal\_pkey\_attribute\_t \*attributes,

const unsigned attributes\_len,

unsigned \*state,

hal\_uuid\_t \*result,

unsigned \*result\_len,

const unsigned result\_max,

const hal\_uuid\_t \* const previous\_uuid);

Matches a private key.

extern hal\_error\_t hal\_rpc\_pkey\_set\_attributes(const hal\_pkey\_handle\_t pkey,

const hal\_pkey\_attribute\_t \*const attributes,

const unsigned attributes\_len);

Sets key attributes.

extern hal\_error\_t hal\_rpc\_pkey\_get\_attributes(const hal\_pkey\_handle\_t pkey,

hal\_pkey\_attribute\_t \*attributes,

const unsigned attributes\_len,

uint8\_t \*attributes\_buffer,

const size\_t attributes\_buffer\_len);

Gets key attributes.

extern hal\_error\_t hal\_rpc\_pkey\_export(const hal\_pkey\_handle\_t pkey,

const hal\_pkey\_handle\_t kekek,

uint8\_t \*pkcs8, size\_t \*pkcs8\_len, const size\_t pkcs8\_max,

uint8\_t \*kek, size\_t \*kek\_len, const size\_t kek\_max);

Exports a private key.

extern hal\_error\_t hal\_rpc\_pkey\_import(const hal\_client\_handle\_t client,

const hal\_session\_handle\_t session,

hal\_pkey\_handle\_t \*pkey,

hal\_uuid\_t \*name,

const hal\_pkey\_handle\_t kekek,

const uint8\_t \* const pkcs8, const size\_t pkcs8\_len,

const uint8\_t \* const kek, const size\_t kek\_len,

const hal\_key\_flags\_t flags);

Imports a private key.

extern hal\_error\_t hal\_rpc\_client\_init(void);

Initializes the transport mechanism for RPC client communication to the RPC server. In socket mode, this will create an PF\_UNIX socket to the multiplexer daemon. In serial mode, this will open a serial connection to the HSM.

extern hal\_error\_t hal\_rpc\_client\_close(void);

Closes the transport mechanism for the RPC client to the RPC server.

extern hal\_error\_t hal\_rpc\_server\_init(void);

Iniitializes the RPC server to listen for connections and messages from a RPC client.

extern hal\_error\_t hal\_rpc\_server\_close(void);

Closes the transport mechanism for RPC messages for the server.

extern hal\_error\_t hal\_rpc\_server\_dispatch(const uint8\_t \* const ibuf, const size\_t ilen,

uint8\_t \* const obuf, size\_t \* const olen);

Takes a complete RPC message and calls the appropriate procedure. Contains procedures that unpack RPC message parameters and then calls the actual procedures in sw/libhal/rpc\_api.c

* + - 1. ‘sw/libhal/rpc\_api.c’

This file contains the implementation to all of the RPC functions defined in ‘sw/libhal/hal.h’. These methods are called directly in both the client (ex. libcryptech-pkcs11.so) and the server (ex. STM32 firmware) code, but based on how the project is configure at ‘make’, the behavior changes. The procedures in this class will call either local or remote implementations. The following three types are defined in sw/libhal/hal\_internal.h

* hal\_rpc\_misc\_dispatch\_t
* hal\_rpc\_hash\_dispatch\_t
* hal\_rpc\_pkey\_dispatch\_t

These types contain function pointers to the actual method. Depending on how the project is compiled, it will use either local or remote versions of the functions. The local versions run on the server and complete immediately. The remote versions generate RPC messages, encodes them using SLIP, and sends them to the firmware either directly over serial or via the daemon (cryptech\_muxd) using a PF\_UNIX socket.

* + - 1. ‘sw/libhal/rpc\_server.c’

‘rpc\_server.’ has the code for dispatching RPC request. It doesn't read from the UART. That is done in 'sw/stm32/projects/hsm/hsm.c'. This code resides in the firmware

**Key Functions:**

* hal\_rpc\_server\_init
* hal\_rpc\_server\_close
* hal\_rpc\_server\_dispatch - Takes a complete RPC message and calls the appropriate procedure. Contains procedures that unpack RPC message parameters and then calls the actual procedures in ‘sw/libhal/rpc\_api.c’.
  + - 1. ‘sw/libhal/rpc\_client\_daemon.c’

Socket implementation of RPC transport that's meant to be used with 'cryptech\_muxd'

Connects to a RPC Server using a AF\_UNIX socket and sends the data using SLIP. This is for connecting to 'cryptech\_muxd'

hal\_rpc\_client\_transport\_init()

hal\_rpc\_client\_transport\_close()

hal\_rpc\_send - sends data using SLIP (hal\_slip\_send 🡪 ‘sw/libhal/slip.c’ )

hal\_rpc\_recv- receives data using SLIP (hal\_slip\_recv 🡪 ‘sw/libhal/slip.c’)

‘

hal\_serial\_send\_char - used to send a char over a socket (Uses PF\_UNIX sockets not serial, but name needs to match SLIP implementation)

hal\_serial\_recv\_char - used to receive a char over a socket (Uses PF\_UNIX sockets not serial, but name needs to match SLIP implementation)

* + - 1. ‘sw/libhal/rpc\_client\_serial’

Serial implemetnation of RPC transports that's meant for direct communication with RPC over USB

Serial implementation of hal\_serial\_send\_char and hal\_serial\_recv\_char are in 'sw/libhal/rpc\_serial.c'

hal\_rpc\_client\_transport\_init()

hal\_rpc\_client\_transport\_close()

hal\_rpc\_send - sends data using SLIP (hal\_slip\_send 🡪 ‘sw/libhal/slip.c’ )

hal\_rpc\_recv- receives data using SLIP (hal\_slip\_recv 🡪 ‘sw/libhal/slip.c’)

* + - 1. Implementation of actual RPC commands in Firmware

On a client, the RPC functions generate a SLIP message that can be sent either over a socket or serial, but in the firmware (server), the method needs to actually be done. The server side implementation has been split into multiple files based on the type of work the function call does.

* ‘sw/libhal/rpc\_hash.c’
* ‘sw/libhal/rpc\_misc.c’
* ‘sw/libhal/rpc\_pkcs1.c’
* ‘sw/libhal/rpc\_pkey.c’
  + 1. Repository README from CrypTech

## Overview ##

This library combines a set of low-level API functions which talk to the Cryptech FPGA cores with a set of higher-level functions providing various cryptographic services.

There's some overlap between the low-level code here and the low-level code in core/platform/novena, which will need sorting out some day, but at the time this library forked that code, the core/platform/novena code was all written to support a test harness rather than a higher-level API.

Current contents of the library:

* Low-level I/O code (FMC, EIM, and I2C).
* An implementation of AES Key Wrap using the Cryptech AES core.
* An interface to the Cryptech CSPRNG.
* An interface to the Cryptech hash cores, including HMAC.
* An implementation of PBKDF2.
* An implementation of RSA, optionally using the Cryptech ModExp core.
* An implementation of ECDSA, optionally using the Cryptech ECDSA base point multiplier cores.
* An interface to the Master Key Memory interface core on the Cryptech Alpha platform.
* A simple keystore implementation with drivers for RAM and flash storage on the Cryptech Alpha platform.
* A remote procedure call (RPC) interface.
* (Just enough) ASN.1 code to support a uniform interface to public (SubjectPublicKeyInformation (SPKI)) and private (PKCS #8) keys.
* A simple key backup mechanism, including a Python script to drive it from the client side.
* An RPC multiplexer to allow multiple clients (indepedent processes) to talk to the Cryptech Alpha at once.
* Client implenetations of the RPC mechanism in both C and Python.
* Test code for all of the above.

Most of these are fairly well self-contained, although the PBKDF2 implementation uses the hash-core-based HMAC implementation with fallback to a software implementation if the cores aren't available.

The major exceptions are the RSA and ECDSA implementations, which uses an external bignum implementation (libtfm) to handle a lot of the arithmetic. In the long run, much or all of this may end up being implemented in Verilog, but for the moment all of the RSA math except for modular exponentiation is happening in software, as is all of the math for ECDSA verification; ECDSA math for key generation and signing on the P-256 and P-384 curves is done in the ECDSA base point

multiplier cores when those are available.

## RSA ##

The RSA implementation includes a compile-time option to bypass the ModExp core and do everything in software, because the ModExp core is a tad slow at the moment (others are hard at work fixing this).

The RSA implementation includes optional blinding (enabled by default).

## ECDSA ##

The ECDSA implementation is specific to the NIST prime curves P-256, P-384, and P-521.

The ECDSA implementation includes a compile-time option to allow test code to bypass the CSPRNG in order to test against known test vectors. Do \*\*NOT\*\* enable this in production builds, as ECDSA depends on good random numbers not just for private keys but for individual signatures, and an attacker who knows the random number used for a particular signature can use this to recover the private key. Arguably, this option should be removed from the code entirely.

The ECDSA software implementation attempts to be constant-time, to reduce the risk of timing channel attacks. The algorithms chosen for the point arithmetic are a tradeoff between speed and code complexity, and can probably be improved upon even in software; reimplementing at

least the field arithmetic in hardware would probably also help. Signing and key generation performance is significantly better when the ECDSA base point multiplier cores are available.

The point addition and point doubling algorithms in the current ECDSA software implementation come from the [EFD][]. At least at the moment, we're only interested in ECDSA with the NIST prime curves, so we use algorithms optimized for a=-3.

The point multiplication algorithm is a straightforward double-and-add loop, which is not the fastest possible algorithm, but is relatively easy to confirm by inspection as being constant-time within the limits imposed by the NIST curves. Point multiplication could probably be made faster by using a non-adjacent form (NAF) representation for the scalar, but the author doesn't understand that well enough to

implement it as a constant-time algorithm. In theory, changing to a NAF representation could be done without any change to the public API.

Points stored in keys and curve parameters are in affine format, but point arithmetic is performed in Jacobian projective coordinates, with the coordinates themselves in Montgomery form; final mapping back to affine coordinates also handles the final Montgomery reduction.

## Keystore ##

The keystore is basically a light-weight database intended to be run directly over some kind of block-access device, with an internal low-level driver interface so that we can use the same API for

multiple keystore devices (eg, flash for "token objects" and RAM for "session objects", in the PKCS #11 senses of those terms).

The available storage is divided up into "blocks" of a fixed size; for simplicity, the block size is a multiple of the subsector size of the flash chip on the Alpha platform, since that's the minimum erasable

unit. All state stored in the keystore itself follows the conventions needed for flash devices, whether the device in question is flash or not. The basic rule here is that one can only clear bits, never set

them: the only way to set a bit is to erase the whole block and start over. So blocks progress from an initial state ("erased") where all bits are set to one, through several states where the block contains

useful data, and ending in a state where all bits are set to zero ("zeroed"), because that's the way that flash hardware works.

The keystore implementation also applies a light-weight form of wear leveling to all keystore devices, whether they're flash devices or not. The wear-leveling mechanism is not particularly sophisticated,

but should suffice. The wear-leveling code treats the entirety of a particular keystore device as a ring buffer of blocks, and keeps track of which blocks have been used recently by zeroing blocks upon freeing them rather than erasing them immediately, while also always keeping the block at the current head of the free list in the erased state. Taken together, this is enough to recover location of the block at the head of the free list after a reboot, which is sufficient for a round-robin wear leveling strategy.

The block format includes a field for a CRC-32 checksum, which covers the entire block except for a few specific fields which need to be left out. On reboot, blocks with bad CRC-32 values are considered

candidates for reuse, but are placed at the end of the free list, preserve their contents for as long as possible in case the real problem is a buggy firmware update.

At the moment, the decision about whether to use the CRC-32 mechanism is up to the individual driver: the flash driver uses it, the RAM driver (which never stores anything across reboots anyway) does not.

Since the flash-like semantics do not allow setting bits, updates to a block always consist of allocating a new block and copying the modified data. The keystore code uses a trivial lock-step protocol

for this: first:

1. The old block is marked as a "tombstone";
2. The new block (with modified data) is written;
3. The old block is erased.

This protocol is deliberately as simple as possible, so that there is always a simple recovery path on reboot.

Active blocks within a keystore are named by UUIDs. With one exception, these are always type-4 (random) UUIDs, generated directly from output of the TRNG. The one exception is the current PIN block, which always uses the reserved all-zeros UUID, which cannot possibly conflict with a type-4 UUID (by definition).

The core of the keystore mechanism is the `ks->index[]` array, which contains nothing but a list of block numbers. This array is divided into two parts: the first part is the index of active blocks, which is

kept sorted (by UUID); the second part is the round-robin free list. Everything else in the keystore is indexed by these block numbers, which means that the index array is the only data structure which the

keystore code needs to sort or rotate when adding, removing, or updating a block. Because the block numbers themselves are small integers, the index array itself is small enough that shuffling data

within it using `memmove()` is a relatively cheap operation, which in turn avoids a lot of complexity that would be involved in managing more sophisticated data structures.

The keystore code includes both caching of recently used keystore blocks (to avoid unnecessary flash reads) and caching of the location of the block corresponding to a particular UUID (to avoid unnecessary index searches). Aside from whatever direct performance benefits this

might bring, this also frees the pkey layer that sits directly on top of the keystore code from needing to keep a lot of active state on particular keystore objects, which is important given that this whole

thing sits under an RPC protocol driven by a client program which can impose arbitrary delays between any two operations at the pkey layer.

## Key backup ##

The key backup mechanism is a straightforward three-step process, mediated by a Python script which uses the Python client implementation of the RPC mechanism. Steps:

1. Destination HSM (target of key transfer) generates an RSA keypair, exports the public key (the "key encryption key encryption key" or "KEKEK").
2. Source HSM (origin of the key transfer) wraps keys to be backed up using AES keywrap with key encryption keys (KEKs) generated by the TRNG; these key encryption keys are in turn encrypted with RSA public key (KEKEK) generated by the receipient HSM.
3. Destination HSM receives wrapped keys, unwraps the KEKs using the KEKEK then unwraps the wrapped private keys.

Transfer of the wrapped keys between the two HSMs can be by any convenient mechanism; for simplicity, `cryptech\_backup` script bundles everything up in a text file using JSON and Base64 encoding.

## Multiplexer daemon ##

While the C client library can be built to talk directly to the Cryptech Alpha board, in most cases it is more convenient to use the `cryptech\_muxd` multiplexer daemon, which is now the default. Client

code talks to `cryptech\_muxd` via a `PF\_UNIX` socket; `cryptech\_muxd` handles interleaving of messages between multiple clients, and also manages access to the Alpha's console port.

The multiplexer requires two external Python libraries, Tornado (version 4.0 or later) and PySerial (version 3.0 or later).

In the long run, the RPC mechanism will need to be wrapped in some kind of secure channel protocol, but we're not there yet.

– sw/libhal/README.md

* 1. stm32

The stm32 repository contains the source code for the firmware. It uses libhal for its implementation of the various cryptographic functions as well as for the RPC server code. This repository also includes the code for the bootloader.

<https://git.cryptech.is/sw/stm32.git>

* + 1. Firmware Details

The main module for the firmware is located in ‘sw/stm32/projects/hsm/hsm.c’. ‘main’ sets up the worker task that will be used to handle RPC request and command line interface (CLI) activity. There is a single thread for each.

* + - 1. RPC Request

RPC request are received and the RPC (user) USB port on the HSM. These are the basic steps used to set up the RPC server and for handling incoming RPC messages:

main() 🡪 ‘sw/stm32/projects/hsm/hsm.c’

1. RPC server is initialized with call to ‘hal\_rpc\_server\_int’ (‘sw/libhal/rpc\_server.c’)
2. Create worker task (dispatch\_task) to handle incoming messages
3. Start UART receiver to receive data into a circular DMA buffer.
   1. The callback to receive the data is RxCallback (‘sw/stm32/projects/hsm/hsm.c’)
      1. Process new character read and add it to the input buffer (ibuf).
         1. The character is processed using hal\_slip\_process\_char (‘sw/libhal/slip.c’)
      2. Once a completed message has been received
         1. Activate a thread to handle the request using a dispatch\_task if available.
            1. Call ‘hal\_rpc\_server\_dispatch’ (‘sw/libhal/rpc\_server.c’ to make the remote procedure call.
         2. If a dispatch\_task is not available, wake the busy task to retry scheduling a dispatch task.
         3. Command Line Interface

The command line interface (CLI) uses the CTY (MGMT) USB port on the HSM. It uses the third-party library ‘libcli’. ‘libcli’ is by David Parrish and according to the description on GitHub, “provides a shared library for including a Cisco-like command-line interface into other software. It’s a telnet interface which supports command-line editing, history, authentication and callbacks for a user-definable function tree.”

The main function for CLI is ‘cli\_main’ and is implemented in ‘mgmt-cli.c’. These are the basics steps used to set up the CLI:

cli\_main() 🡪 ‘sw/stm32/projects/hsm/mgmt.\_cli.c’

1. Initialize ‘libcli’
   1. ‘cli\_init’
2. Set primary CLI callbacks
   1. ‘cli\_read\_callback(cli, uart\_cli\_read)’
   2. ‘cli\_write\_callback(cli, uart\_cli\_write)’
   3. ‘cli\_print\_callback(cli, uart\_cli\_print)’
3. Set banner and host name
   1. ‘cli\_set\_banner(cli, “Cryptech Alpha”)’
   2. ‘cli\_set\_hostname(cli, “cryptech”)
   3. ‘cli\_set\_auth\_callback(cli, check\_auth)
4. Add configurations for FPGA, keystore, master key, firmware, bootloader, and misc commands.
5. Start CLI loop
   1. ‘cli\_loop(cli, 0)’
      1. Repository README from CrypTech

STM32 firmware for Cryptech Alpha board

=======================================

The Alpha board is our first full prototype for an open-source hardware security module (HSM). It is a custom board with an STM32 Cortex-M4 microcontroller and an Artix-7 FPGA, flash-based keystore, separate memory for the Key Encryption Key, etc. See the `hardware` repository for schematics and production files. See the wiki for design documents.

The code in this repository builds the firmware that provides the HSM functionality on the Alpha board.

There is some residual code here to support the "dev-bridge" board, a daughterboard for the Novena, which talks to the Novena's FPGA through the high-speed expansion connector. Only a few of these boards were ever made,

and all development/testing ceased as soon as the Alpha became available, so the dev-bridge should be considered deprecated, and support may be removed in the future.

Copyrights

==========

The license for all work done on this in the CrypTech project is a 3-clause BSD license.

Third-party components, as well as code generated using the STMicroelectronics initialization code generator STM32CubeMX, or adapted from STM example/support code, may have different licensing, detailed

below.

Components

==========

Libraries

---------

* `mbed` - A stripped down copy of the ARM CMSIS library, copied from the mbed github (see `libraries/mbed/README.txt` for details). The bulk of this library is covered under 3-clause BSD licenses from either ARM or STMicroelectronics, but one file is covered under an Apache license from ARM.
* `libhal` - Build directory for our own Hardware Adaption Library (hardware-independent Cryptech components). Source is expected to be in `sw/libhal`.
* `libtfm` - Build directory for "Tom's Fast Math", which is used heavily for bignum math in the RSA and ECDSA code. This code is covered under an unrestricted public domain license, and source is expected to be in `sw/thirdparty/libtfm`.
* `libcli` - Build directory for a third-party Command Line Interface library. The source is not currently under `sw/thirdparty` because the license is LGPLv2.1; we are negotiating to see if we can get a BSD-compatible license for it.
* `libprof` - A port of the `gmon` profiling package, to be used in development only, not in production code (obviously). The licensing is a mix of BSD and "Cygwin license", which now seems to be LGPLv3.

Projects

--------

These directories build different firmware images for the Alpha board.

* `hsm` - Firmware providing HSM functionality. Clients communicate via RPC requests on the USER USB port, or interactively on the MGMT USB port.
* `bootloader` - The first thing that runs on the device. It either starts the primary firmware, or installs new firmware.
* `board-test` - Tests of hardware components.
* `cli-test` - Test of the CLI itself, plus some interactive tests of hardware components. Duplicates way too much of the HSM CLI.
* `libhal-test` - A framework for running the libhal component tests. Hasn't been run in a while, probably still works.

Building

========

Our primary build environments are Debian and Ubuntu, but this should work on any system with Gnu tools installed.

The following packages need to be installed:

$ apt-get install gcc-arm-none-eabi gdb-arm-none-eabi openocd

The Makefile assumes that all Cryptech repositories have been fetched into a canonical directory structure, e.g. `libhal` and `thirdparty` are siblings to this directory, under `sw`.

To build the source code, issue `make` from the top level directory (where this file is). The first time, this will build the complete STM CMSIS library. A subsequent `make clean` will \*not\* clean away the CMSIS

library, but a `make distclean` will.

Installing

==========

Do `bin/flash-target` from the top level directory (where this file is) to flash a built image into the microcontroller. See the section ST-LINK below for information about the actual hardware programming device needed.

Example loading the HSM firmware:

$ make hsm

$ ./bin/flash-target projects/hsm/hsm

At this point, the STM32 will reset into the bootloader which flashes the blue LED five times in one second, and then jumps to the primary firmware.

Once the bootloader is installed, regular firmware can be loaded without an ST-LINK cable like this:

$ cryptech\_upload --firmware -i projects/hsm/hsm.bin

Then reboot the Alpha board.

ST-LINK

-------

To program the MCU, an ST-LINK adapter is used. The cheapest way to get one is to buy an evaluation board with an ST-LINK integrated, and pinouts to program external chips. This should work with any evaluation board from

STM; we have tested with STM32F4DISCOVERY (with ST-LINK v2.0) and NUCLEO-F411RE (with ST-LINK v2.1).

The ST-LINK programming pins is called J1 and is near the CrypTech logo printed on the circuit board. The pin-outs is shown on the circuit board (follow the thin white line from J1 to the white box with STM32\_SWD

written in it). From left to right, the pins are

3V3, CLK, GND, I/O, NRST and N/C

This matches the pin-out on the DISCO and NUCLEO boards we have tried.

First remove the pair of ST-LINK jumpers (CN4 on the DISCO, CN2 on the NUCLEO). Then find the 6-pin SWD header on the left of the STM board (CN2 on the DISCO, CN4 on the NUCLEO), and connect them to the Alpha board:

NUCLEO / DISCO CRYPTECH ALPHA

-------------- --------------

\* 1 VDD\_TARGET <-> 3V3

\* 2 SWCLK / T\_JTCK <-> CLK

\* 3 GND <-> GND

\* 4 SWDIO / T\_JTMS <-> IO

\* 5 NRST / T\_NRST <-> NRST

\* 6 N/C

The Alpha board should be powered on before attempting to flash it.

Debugging the firmware

----------------------

[This site](http://fun-tech.se/stm32/OpenOCD/gdb.php) shows several ways to use various debuggers to debug the firmware in an STM32.

There is a shell script called 'bin/debug' that starts an OpenOCD server and GDB. Example:

$ ./bin/debug projects/hsm/hsm

Once in GDB, issue `monitor reset halt` to reset the STM32 before debugging.

Remember that the first code to run will be the bootloader, but if you do e.g. `break main` and `continue` you will end up in main() after the bootloader has jumped there.

– sw/stm32/README.md

* 1. pkcs11

PKCS#11 is the standard used to access the alpha boards cryptography functions to outside applications such as BIND and OpenDNSSEC.

<https://git.cryptech.is/sw/pkcs11.git>

* + 1. Implemented PKCS#11 Functions

CK\_RV C\_Initialize(CK\_VOID\_PTR pInitArgs)

Initializes Cryptoki

CK\_RV C\_Finalize(CK\_VOID\_PTR pReserved)

Clean up miscellaneous Cryptoki-associated resources

CK\_RV C\_GetFunctionList(CK\_FUNCTION\_LIST\_PTR\_PTR ppFunctionList)

Obtains entry points of Cryptoki library functions

CK\_RV C\_GetSlotList(CK\_BBOOL tokenPresent,

CK\_SLOT\_ID\_PTR pSlotList,

CK\_ULONG\_PTR pulCount)

Obtains a list of slots in the system.

CK\_RV C\_GetTokenInfo(CK\_SLOT\_ID slotID,

CK\_TOKEN\_INFO\_PTR pInfo)

Obtains information about a particular token.

CK\_RV C\_OpenSession(CK\_SLOT\_ID slotID,

CK\_FLAGS flags,

CK\_VOID\_PTR pApplication,

CK\_NOTIFY Notify,

CK\_SESSION\_HANDLE\_PTR phSession)

Opens a connection between an application and a particular token or sets up an application callback for token insertion.

CK\_RV C\_CloseSession(CK\_SESSION\_HANDLE hSession)

Closes a session.

CK\_RV C\_CloseAllSessions(CK\_SLOT\_ID slotID)

Closes all sessions with a token

CK\_RV C\_Login(CK\_SESSION\_HANDLE hSession,

CK\_USER\_TYPE userType,

CK\_UTF8CHAR\_PTR pPin,

CK\_ULONG ulPinLen)

Logs into a token.

CK\_RV C\_Logout(CK\_SESSION\_HANDLE hSession)

Logs out from a token.

CK\_RV C\_CreateObject(CK\_SESSION\_HANDLE hSession,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount,

CK\_OBJECT\_HANDLE\_PTR phObject)

Creates an object.

CK\_RV C\_DestroyObject(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hObject)

Destroys an object.

CK\_RV C\_GetAttributeValue(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hObject,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount)

Obtains an attribute value of an object.

CK\_RV C\_FindObjectsInit(CK\_SESSION\_HANDLE hSession,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount)

Initializes an object search operation.

CK\_RV C\_FindObjects(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE\_PTR phObject,

CK\_ULONG ulMaxObjectCount,

CK\_ULONG\_PTR pulObjectCount)

Continues an object search operation.

CK\_RV C\_FindObjectsFinal(CK\_SESSION\_HANDLE hSession)

Finishes an object search operation.

CK\_RV C\_DigestInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism)

Initializes a message-digesting operation.

CK\_RV C\_Digest(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pDigest,

CK\_ULONG\_PTR pulDigestLen)

Digests single-part data.

CK\_RV C\_DigestUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen)

Continues a multiple-part digesting operation.

CK\_RV C\_DigestFinal(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pDigest,

CK\_ULONG\_PTR pulDigestLen)

Finishes a multiple-part digesting operation.

CK\_RV C\_SignInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey)

Initializes a signature operation.

CK\_RV C\_Sign(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG\_PTR pulSignatureLen)

Signs single-part data.

CK\_RV C\_SignUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen)

Continues a multiple-part signature operation.

CK\_RV C\_SignFinal(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pSignature,

CK\_ULONG\_PTR pulSignatureLen)

Finishes a multiple-part signature operation.

CK\_RV C\_VerifyInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey )

Initializes a verification operation.

CK\_RV C\_Verify(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG ulSignatureLen)

Verifies a signature on single-part data.

CK\_RV C\_VerifyUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen)

Continues a multiple-part verification operation.

CK\_RV C\_VerifyFinal(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pSignature,

CK\_ULONG ulSignatureLen)

Finishes a multiple-part verification operation.

CK\_RV C\_GenerateKeyPair(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_ATTRIBUTE\_PTR pPublicKeyTemplate,

CK\_ULONG ulPublicKeyAttributeCount,

CK\_ATTRIBUTE\_PTR pPrivateKeyTemplate,

CK\_ULONG ulPrivateKeyAttributeCount,

CK\_OBJECT\_HANDLE\_PTR phPublicKey,

CK\_OBJECT\_HANDLE\_PTR phPrivateKey)

Generates a public-key/private-key pair.

CK\_RV C\_GenerateRandom(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR RandomData,

CK\_ULONG ulRandomLen)

Generates random data.

CK\_RV C\_GetMechanismInfo(CK\_SLOT\_ID slotID,

CK\_MECHANISM\_TYPE type,

CK\_MECHANISM\_INFO\_PTR pInfo)

Obtains information about a particular mechanism.

CK\_RV C\_GetSessionInfo(CK\_SESSION\_HANDLE hSession,

CK\_SESSION\_INFO\_PTR pInfo)

Otains information about the session.

CK\_RV C\_GetInfo(CK\_INFO\_PTR pInfo)

Obtains general information about Cryptoki.

CK\_RV C\_GetSlotInfo(CK\_SLOT\_ID slotID,

CK\_SLOT\_INFO\_PTR pInfo)

Obtains information about a particular slot.

CK\_RV C\_GetMechanismList(CK\_SLOT\_ID slotID,

CK\_MECHANISM\_TYPE\_PTR pMechanismList,

CK\_ULONG\_PTR pulCount)

Obtains a list of mechanisms supported by a token.

CK\_RV C\_SeedRandom(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pSeed,

CK\_ULONG ulSeedLen)

Mixes in additional seed material to the random number generator.

CK\_RV C\_GetFunctionStatus(CK\_SESSION\_HANDLE hSession)

Legacy function which always returns ‘CKR\_FUNCTION\_NOT\_PARALLEL’.

CK\_RV C\_CancelFunction(CK\_SESSION\_HANDLE hSession)

Legacy function which always returns ‘CKR\_FUNCTION\_NOT\_PARALLEL’.

* + 1. Unimplemented PKCS#11 Functions

The following is a list of unsupported functions. This functions have been implemented as stubs in the code and all return ‘CKR\_FUNCTION\_NOT\_SUPPORTED’.

CK\_RV C\_GenerateKey(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount,

CK\_OBJECT\_HANDLE\_PTR phKey)

Generates a secret key.

CK\_RV C\_InitToken(CK\_SLOT\_ID slotID,

CK\_UTF8CHAR\_PTR pPin,

CK\_ULONG ulPinLen,

CK\_UTF8CHAR\_PTR pLabel)

Initializes a token.

CK\_RV C\_InitPIN(CK\_SESSION\_HANDLE hSession,

CK\_UTF8CHAR\_PTR pPin,

CK\_ULONG ulPinLen)

Initializes the normal user’s PIN.

CK\_RV C\_SetPIN(CK\_SESSION\_HANDLE hSession,

CK\_UTF8CHAR\_PTR pOldPin,

CK\_ULONG ulOldLen,

CK\_UTF8CHAR\_PTR pNewPin,

CK\_ULONG ulNewLen)

Modifies the PIN of the current user.

CK\_RV C\_GetOperationState(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pOperationState,

CK\_ULONG\_PTR pulOperationStateLen)

Obtains the cryptographics operations state of a session.

CK\_RV C\_SetOperationState(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pOperationState,

CK\_ULONG ulOperationStateLen,

CK\_OBJECT\_HANDLE hEncryptionKey,

CK\_OBJECT\_HANDLE hAuthenticationKey)

Sets the cryptographics operations state of a session.

CK\_RV C\_CopyObject(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hObject,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount,

CK\_OBJECT\_HANDLE\_PTR phNewObject)

Creates a copy of an object.

CK\_RV C\_GetObjectSize(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hObject,

CK\_ULONG\_PTR pulSize)

Obtains the size of an object in bytes.

CK\_RV C\_SetAttributeValue(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hObject,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulCount)

Modifies an attribute value of an object.

CK\_RV C\_EncryptInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey)

Initializes an encryption operation.

CK\_RV C\_Encrypt(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pEncryptedData,

CK\_ULONG\_PTR pulEncryptedDataLen)

Encrypts single-part data.

CK\_RV C\_EncryptUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG\_PTR pulEncryptedPartLen)

Continues a multiple-part encryption operation.

CK\_RV C\_EncryptFinal(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pLastEncryptedPart,

CK\_ULONG\_PTR pulLastEncryptedPartLen)

Finishes a multiple-part encryption operation.

CK\_RV C\_DecryptInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey)

Initializes a decryption operation.

CK\_RV C\_Decrypt(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pEncryptedData,

CK\_ULONG ulEncryptedDataLen,

CK\_BYTE\_PTR pData,

CK\_ULONG\_PTR pulDataLen)

Decrypts single-part encrypted data.

CK\_RV C\_DecryptUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG ulEncryptedPartLen,

CK\_BYTE\_PTR pPart,

CK\_ULONG\_PTR pulPartLen)

Continues a multiple-part decryption operation.

CK\_RV C\_DecryptFinal(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pLastPart,

CK\_ULONG\_PTR pulLastPartLen)

Finishes a multiple-part decryption operation.

CK\_RV C\_DigestKey(CK\_SESSION\_HANDLE hSession,

CK\_OBJECT\_HANDLE hKey)

Digest a key.

CK\_RV C\_SignRecoverInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey)

Initializes a signature operation, where the data can be recovered from the signature.

CK\_RV C\_SignRecover(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG\_PTR pulSignatureLen)

Signs single-part data, where the data can be recovered from the signature.

CK\_RV C\_VerifyRecoverInit(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey)

Initializes a verification operation where the data is recovered from the signature.

CK\_RV C\_VerifyRecover(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pSignature,

CK\_ULONG ulSignatureLen,

CK\_BYTE\_PTR pData,

CK\_ULONG\_PTR pulDataLen)

Verifies a signature on single-part data, where the data is recovered from the signature.

CK\_RV C\_DigestEncryptUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG\_PTR pulEncryptedPartLen)

Continues simultaneous multiple-part digesting and encryption operations.

CK\_RV C\_DecryptDigestUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG ulEncryptedPartLen,

CK\_BYTE\_PTR pPart,

CK\_ULONG\_PTR pulPartLen)

Continues simultaneous multi-part decryption and digesting operations.

CK\_RV C\_SignEncryptUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pPart,

CK\_ULONG ulPartLen,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG\_PTR pulEncryptedPartLen)

Continues simultaneous multiple-part signature and encryption operations.

CK\_RV C\_DecryptVerifyUpdate(CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pEncryptedPart,

CK\_ULONG ulEncryptedPartLen,

CK\_BYTE\_PTR pPart,

CK\_ULONG\_PTR pulPartLen)

Continues simultaneous multiple-part decryption and verification operations

CK\_RV C\_WrapKey(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hWrappingKey,

CK\_OBJECT\_HANDLE hKey,

CK\_BYTE\_PTR pWrappedKey,

CK\_ULONG\_PTR pulWrappedKeyLen)

Wraps (encrypts) a key.

CK\_RV C\_UnwrapKey(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hUnwrappingKey,

CK\_BYTE\_PTR pWrappedKey,

CK\_ULONG ulWrappedKeyLen,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulAttributeCount,

CK\_OBJECT\_HANDLE\_PTR phKey)

Unwraps (decrypts) a key.

CK\_RV C\_DeriveKey(CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hBaseKey,

CK\_ATTRIBUTE\_PTR pTemplate,

CK\_ULONG ulAttributeCount,

CK\_OBJECT\_HANDLE\_PTR phKey)

Derives a key from a base key.

CK\_RV C\_WaitForSlotEvent(CK\_FLAGS flags,

CK\_SLOT\_ID\_PTR pSlot,

CK\_VOID\_PTR pRserved)

Waits for a slot event (token insertion, removal, etc.) to occur.

* + 1. Repository README from CrypTech

## Introduction ##

This is an implementation of the [PKCS11][] API for the [Cryptech][] project. Like most PKCS #11 implementations, this one is incomplete and probably always will be: PKCS #11 is very open-ended, and the specification includes enough rope for an unwary developer to hang not only himself, but all of his friends, relations, and casual acquaintances.

Along with the PKCS #11 library itself, the package includes a companion Python interface ("cryptech.py11"), which uses the ctypes module from the Python standard library to talk to the PKCS #11 implementation. The Python implementation is intended primarily to simplify testing the C code, but can be used for other purposes; while it seems unlikely that anything could ever make PKCS #11 "fun", the `cryptech.py11` library attempts to make it a bit less awful by providing both direct acess to the raw PKCS #11 API and a somewhat more "pythonic" API layered on top of the raw API.

## Novel design features ##

[PKCS11][]'s data model involves an n-level-deep hierarchy of object classes, which is somewhat tedious to implement correctly in C, particularly if one wants the correspondence between specification and code to be at all obvious. In order to automate much of the drudge work involved, this implementation uses an external representation of the object class hierarchy, which is processed at compile time by a Python script to generate tables which drive the C code which performs the necessary type checking.

## Current status ##

As of this writing, the implementation supports only the RSA, ECDSA, SHA-1, and SHA-2 algorithms, but the design is intended to be extensible.

The underlying cryptographic support comes from the [Cryptech][] `libhal` package.

Testing to date has been done using the `bin/pkcs11/` tools from the BIND9 distribution, the `hsmcheck` and `ods-hsmutil` tools from the OpenDNSSEC distribution, the `hsmbully` diagnostic tool, the Google `pkcs11test` test suite, and a somewhat ad hoc set of unit tests using Python's unittest library along with our own `cryptech.py11` library.

The library is also known to work as an `OpenSSL` engine when used with the `engine-pkcs11` package spun out of the OpenSC project. This has not been tested extensively, but key generation, signature, and verification all work (with RSA keys -- the engine appears not to understand ECDSA keys, we have not investigated into details here).

## Copyright status ##

The [PKCS11][] header files are "derived from the RSA Security Inc. PKCS #11 Cryptographic Token Interface (Cryptoki)". See the `pkcs11\*.h` header files for details.

Code written for the [Cryptech][] project is under the usual Cryptech BSD-style license.

[PKCS11]: http://www.cryptsoft.com/pkcs11doc/STANDARD/ "PKCS #11"

[Cryptech]: https://cryptech.is/ "Cryptech"

– sw/pkcs11/README.md

* 1. tamper

This documentation does not examine the code for tamper circuit; however, the README.md file has been included for reference purposes.

<https://git.cryptech.is/sw/tamper.git>

* + 1. Repository README from CrypTech

# Cryptech tamper detection

This is software for the Atmel AVR ATtiny828 MCU on the Cryptech alpha board, rev02, implementing tamper detection and master key erasure.

## Overview

\*\*\*\*\*\*\*\*\*\*\*\*\*

\* P A N I C \*

\* button \*

\*\*\*\*\*\*\*\*\*\*\*\*\*

/

/

/

AVR ---- SPI mux ---- FPGA

| |

| ARM

MKM

AVR -- Atmel MCU

FPGA -- FPGA

MKM -- Master Key Memory, 23K640 SRAM

SPI mux -- 2 x MC74AC244DW

ARM -- ARM CPU

The MKM holds the master key for the device.

The AVR, MKM and the mux are all battery powered.

The AVR and the FPGA are both sharing access to the MKM through the mux, with the AVR connected to the pins used for deciding who's in control of the memory. If the AVR doesn't actively grab control of the MKM, the FPGA is in control.

When the panic button is pressed, the AVR takes control over the MKM and writes zeros to it as quickly as possible. In idle mode, i.e. when the panic button is not pressed, the AVR tries to consume as little power as possible.

## Building the software

To build a .hex file suitible for uploading to a board with a ATTiny828, a C compiler for AVR is needed, as wells a objcopy. On a Debian system, the following command can be used for installing both:

apt-get install gcc-avr binutils-avr avr-libc

To build tamper.hex, type 'make' in this directory.

To upload a .hex file to a board, the program avrdude can be used. On a Debian system, the following command can be used for installing avrdude:

apt-get install avrdude

If configuration for ATtiny828 is missing, the file attiny828.conf in this directory could be appended to avrdude.conf:

cat attiny828.conf >> /etc/avrdude.conf

Often, a piece of hardware called "SPI programmer" is needed in order to upload the .hex file to the target system. The one I've been using has "sparkfun.com" printed on it. This small board has a mini-USB port to connect to a host system and a header with SPI pins to connect to a board with an AVR on it.

To upload a .hex file to a board, use the upload.sh shell script in this directory with the name of the file as the only argument:

./upload.sh tamper.hex

Depending on permissions on your host system you might want to run the

upload script as root.

## GPIO on Cryptech HSM rev.03

The GPIO ports are located on JP5 (AVR\_GPIO). From left to right, as seen when the marking is above the connector, the ports are:

1. 3V3

2. PORTC0

3. PORTC1

4. PORTC2

5. PORTC3

6. PORTC4

7. PORTC5

8. PORTC6

9. PORTC7

10. GND

## Dependencies

### Debian

- apt-get install gcc-avr binutils-avr avr-libc avrdude

### Fedora

- dnf install avrdude avr-gcc avr-libc

– sw/tamper/README.md

1. Key CrypTech Software
   1. ‘cryptech\_probe’ (Python)

The source code for this project can be found at ‘Cryptech/alpha/releng/source/sw/stm32/projects/hsm’.

This is a utility that probes for USB serial ports to see which port the HSM has been plugged into. The output of this utility is sent to stdout so ‘eval $(cryptech\_probe)’ can be used to set the enviroment variables using ‘export’.

**Hardware Information Used in Search**

* VID = 0403
* PID = 6014

**Environment Variables**

* CRYPTECH\_CTY\_CLIENT\_SERIAL\_DEVICE

This environment variable stores the port name for the CTY port.

* CRYPTECH\_RPC\_CLIENT\_SERIAL\_DEVICE

This environment variable stores the port name for the RPC port.

**Probe Finding Algorithm**

1. Get a list of all possible ports using the 0403:6014 VID:PID combination.
2. For each port in the list
   1. Open the port
   2. Send RPC query over the serial port
   3. If result returns ‘Username:’, ‘Password:’, or ‘cryptech>’
      1. Mark as CTY
   4. Else if result is a valid RPC reponse
      1. Mark as RPC
   5. If marked
      1. Set environment variable

**Potential Issues with DKS HSM**

* The current implementation using environment variables could be extended to work with more than one alpha if additional variables are added with indexes. An additional variable with the number of HSM found will also be required to support N alpha boards.
* ‘cryptech\_probe’ only supports one CrypTech alpha. There is no mechanism to distinguish the USB ports if two or more HSMs have been plugged in such as differing serial numbers. While it’s currently possible to programmatically figure out which 2 ports is a pair using the current firmware, it would be better if there was a way to set a serial number on the CTY port and see if the RPC port returns the same value.
  1. ‘cryptech\_muxd’ (Python)

The source code for this project can be found in the ‘Cryptech/alpha/releng/source/sw/libhal’ folder.

‘cryptech\_muxd’ uses the Python PySerial and Tornado libraries for serial and network I/O. It uses a variant on ‘tornado.tcpserver.TCPServer’ to listen on a AF\_UNIX (PF\_LOCAL) socket instead of a TCP socket.

The following PF\_UNIX sockets are used

* env ‘CRYPTECH\_CTY\_CLIENT\_SOCKET\_NAME’

defaults to "/tmp/.cryptech\_muxd.cty"

* env ‘CRYPTECH\_RPC\_CLIENT\_SOCKET\_NAME’

defaults to "/tmp/.cryptech\_muxd.rpc"

* + 1. SerialIOStream (tornado.iostream.BaseIOStream)

This is an implementation of a Tornado IOStream over a PySerial device and is used for communication over serial directly to the HSM. This is used as the base class for communicating on RPC and CTY.

* + 1. PFUnixServer (tornado.tcpserver.TCPServer)

This is a variant on tornado.tcpserver.TCPServer used for listening on a AF\_UNIX (PF\_LOCAL) socket instead of a TCP socket. This is the base class for listening to outside programs that want to send request to the RPC or CTY.

* + 1. RPCIOStream (SerialIOStream)

RPCSIOStream is derived from SerialIOStream and is used to forward formatted RPC request to the HSM over the serial port. This uses SLIP encoding.

* + 1. RPCServer (PFUnixServer)

RPCServer multiplexes Cryptech RPC over a AF\_UNIX socket. Uses the RPCIOStream object that’s passed in at object creation.

**handle\_stream**

1. Read from the stream until the end of a SLIP messsage
2. Decode message
3. Replace client handle in the decoded message
4. Re-encode the message
5. Send the encoded message to the RPC using self.serial.rpc\_input (RPCIOStream object)
6. Encode reply
7. Send encoded reply back over the stream
   * 1. CTYIOStream (SerialIOStream)

This is the Tornado IOStream for a serial console channel and is derived from SerialIOStream. This does not use SLIP encoding.

* + 1. CTYServer (PFUnixServer)

Serves Cryptech console over a PF\_UNIX socket. Uses the CTYIOStream object that’s passed in at object creation. Only one program can communicate over the CTY at a time. If another program tries to connect while a request is being sent to the CTY, the message “[Console already in use, sorry]\n” will be returned.

**handle\_stream**

1. Check to see if CTY is available
2. Set attached\_cty to stream
3. Write bytes from stream to serial
4. Sends result back over the stream
5. clears attached\_cty
   * 1. Start up algorithm
6. If rpc\_device from CRYPTECH\_RPC\_CLIENT\_SERIAL\_DEVICE or args is set
7. Create rpc\_stream (RPCIOStream)
8. Create rpc\_server (RPCServer) and pass in rpc\_stream
9. If cty\_device from CRYPTECH\_CTY\_CLIENT\_SERIAL\_DEVICE or args is set
10. Create cty\_stream (CTYIOStream)
11. Create cty\_server (CTYServer) and pass in cty\_stream
    1. ‘cryptech\_console’ (Python)

The source code for this project can be found in the ‘Cryptech/alpha/releng/source/sw/libhal’ folder. This connects to ‘cryptech\_muxd’ using a AF\_UNIX socket to send and receive information from the CTY. The socket that it will connect to will either be set in the enviroment variable ‘CRYPTECH\_CTY\_CLIENT\_SOCKET\_NAME’ or it will be "/tmp/.cryptech\_muxd.cty" by default.

All console input (stdin) is sent over the AF\_UNIX socket and all data received from the AF\_UNIX socket is written to console output (stdout).

* 1. ‘cryptech\_upload’ (Python)

‘cryptech\_upload’ is a utility for uploading firmware, a bootloader, or a FPGA bitstream to the HSM. This utility can be configured to use cryptech\_muxd as well as a direct USB serial connection. Data is sent of the CTY USB port.

* 1. ‘libhal.py’ (Python)

This is a Python interface to the Cryptech libhal RPC API. This library connects to cryptech\_muxd using a AF\_UNIX socket connection.

**The following methods are available:**

* def get\_version(self):
* def get\_random(self, n):
* def set\_pin(self, user, pin, client = 0):
* def login(self, user, pin, client = 0):
* def logout(self, client = 0):
* def logout\_all(self):
* def is\_logged\_in(self, user, client = 0):
* def hash\_get\_digest\_length(self, alg):
* def hash\_get\_digest\_algorithm\_id(self, alg, max\_len = 256):
* def hash\_get\_algorithm(self, handle):
* def hash\_initialize(self, alg, key = None, client = 0, session = 0, mixed\_mode = None):
* def hash\_update(self, handle, data):
* def hash\_finalize(self, handle, length = None):
* def pkey\_load(self, der, flags = 0, client = 0, session = 0):
* def pkey\_open(self, uuid, client = 0, session = 0):
* def pkey\_generate\_rsa(self, keylen, flags = 0, exponent = "\x01\x00\x01", client = 0, session = 0):
* def pkey\_generate\_ec(self, curve, flags = 0, client = 0, session = 0):
* def pkey\_close(self, pkey):
* def pkey\_delete(self, pkey):
* def pkey\_get\_key\_type(self, pkey):
* def pkey\_get\_key\_curve(self, pkey):
* def pkey\_get\_key\_flags(self, pkey):
* def pkey\_get\_public\_key\_len(self, pkey):
* def pkey\_get\_public\_key(self, pkey, length = None):
* def pkey\_sign(self, pkey, hash = 0, data = "", length = 1024):
* def pkey\_verify(self, pkey, hash = 0, data = "", signature = None):
* def pkey\_match(self, type = 0, curve = 0, mask = 0, flags = 0, attributes = {}, length = 64, client = 0, session = 0):
* def pkey\_set\_attributes(self, pkey, attributes):
* def pkey\_get\_attributes(self, pkey, attributes, attributes\_buffer\_len = 2048):
* def pkey\_export(self, pkey, kekek, pkcs8\_max = 2560, kek\_max = 512):
* def pkey\_import(self, kekek, pkcs8, kek, flags = 0, client = 0, session = 0):
  1. ‘cryptech\_backup’ (Python)

Securely back up private keys from one Cryptech HSM to another.

This works by having the destination HSM (the one importing keys) create an RSA keypair (the "KEKEK"), the public key of which can then be imported into the source HSM (the one exporting keys) and used to encrypt AES key encryption keys (KEKs) which in turn can be used to

wrap the private keys being transfered. Transfers are encoded in JSON; the underlying ASN.1 formats are SubjectPublicKeyInfo (KEKEK public key) and PKCS #8 EncryptedPrivateKeyInfo (everything else).

NOTE WELL: while this process makes it POSSIBLE to back up keys securely, it is not sufficient by itself: the operator MUST make sure only to export keys using a KEKEK known to have been generated by the target HSM. See the unit tests in the source repository for an example of how to fake this in a few lines of Python.

We also implement a software-based variant on this backup mechanism, for cases where there is no second HSM. The protocol is much the same, but the KEKEK is generated in software and encrypted using a symmetric key derived from a passphrase using PBKDF2. This requires the PyCrypto library, and is only as secure as memory on the machine where you're running it (so it's theoretically vulnerable to root or anybody with access to /dev/mem). Don't use this mode unless you understand the risks, and see the "NOTE WELL" above.

# Diagram of the trivial protocol we're using:

#

# SOURCE HSM DESTINATION HSM

#

# Generate and export KEKEK:

# hal\_rpc\_pkey\_generate\_rsa()

# hal\_rpc\_pkey\_get\_public\_key()

#

# Load KEKEK public <--------- Export KEKEK public

# hal\_rpc\_pkey\_load()

# hal\_rpc\_pkey\_export()

#

# Export PKCS #8 and KEK ----------> Load PKCS #8 and KEK, import key

# hal\_rpc\_pkey\_import()

* 1. libcryptech-pkcs11.so (C)

The libcryptech-pkcs11.so library is a shared library that other applications such as BIND and OpenDNSSEC connect to to access the alpha board’s Cryptoki interface. The code is contained in the ‘source/sw/libhal’ and ‘source/sw/libhal/pkcs11’ folders. The library uses the Cryptech RPC API and communicates with the alpha by way of ‘cryptech\_muxd’ using an AF\_UNIX socket.

* + 1. **RPC Setup for communicating through cryptech\_muxd**

RPC setup for communicating through cryptech\_muxd starts in the PKCS#11 function, C\_Initialize which can be found in ‘source/sw/libhal/pkcs11/pkcs11.c’.

→ C\_Initialize() : ‘sw/libhal/pkcs11/pkcs11.c’

* Starts the AF\_UNIX socket connection with ‘cryptech\_muxd’ by calling ‘hal\_rpc\_client\_init’.

→ hal\_rpc\_client\_init() : ‘source/sw/libhal/rpc\_client\_daemon.c’

* Gets the socket name to communicate on from the environment variable, ‘CRYPTECH\_RPC\_CLIENT\_SOCKET\_NAME’.
* Connects to the PF\_UNIX socket and gets a handle to that socket

Once hal\_rpc\_client\_init() returns successfully, all RPC communication is sent using the following functions all inside of ‘sw/libhal/rpc\_client\_daemon.c’:

* ‘hal\_rpc\_client\_transport\_close’
* ‘hal\_serial\_send\_char’
* ‘hal\_serial\_recv\_char’

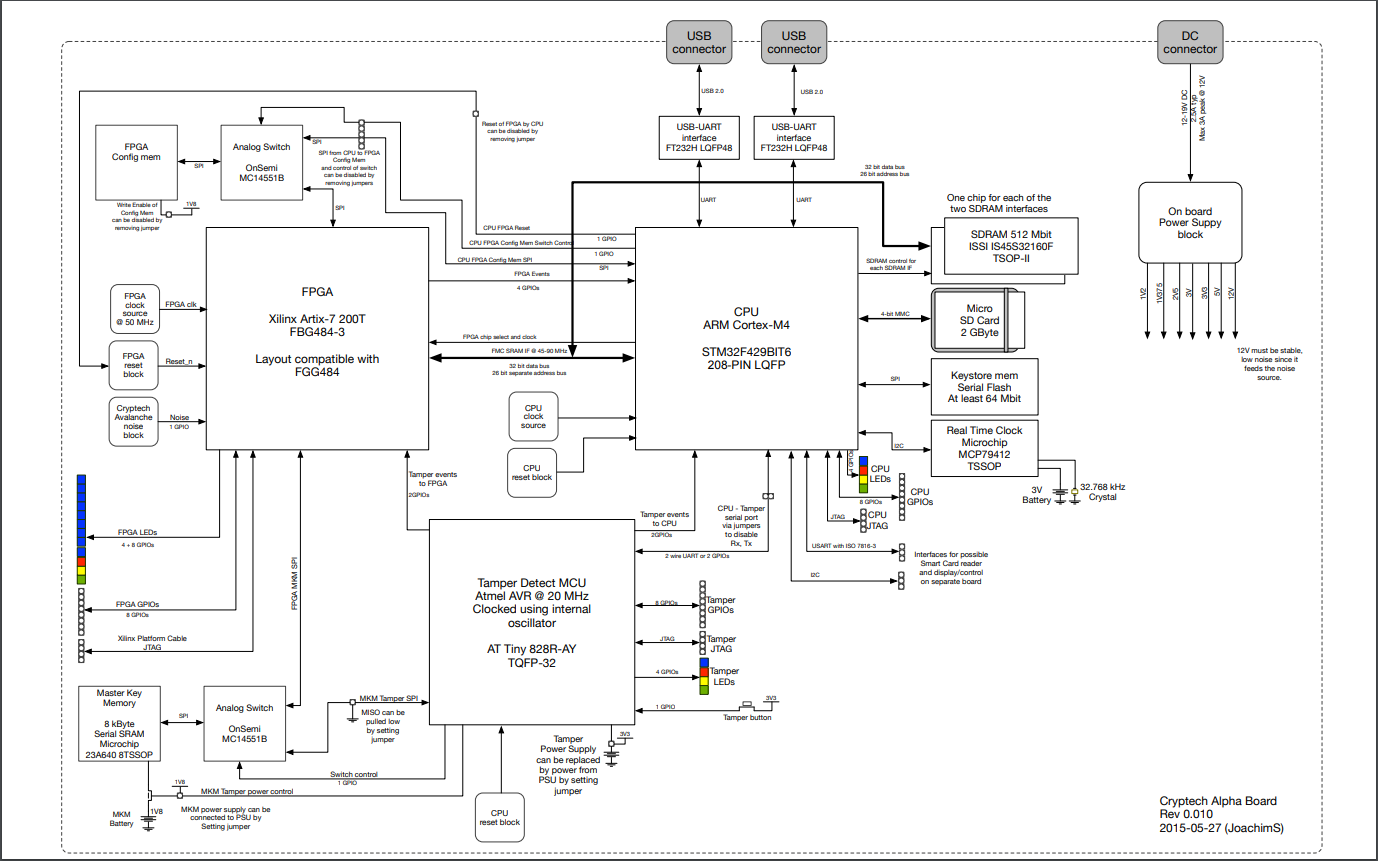
1. CrypTech Software Summary

A screenshot of a computer

Description generated with very high confidence

The CrypTech code communicates to the HSM using two USB ports, CTY and RPC. The RPC port accepts remote procedures calls that are sent as a SLIP encoded byte stream. In most cases, CTY behaves like a terminal and accepts plain text commands, but it is also used for updating the firmware, bootloader, and FPGA. Most applications communicate to the HSM via cryptech\_muxd which uses a tornado server that listens on an AF\_UNIX socket. RPC messages are packaged as XDR (External Data Representation) and always include at least the ID for the command and the sender as well as any parameters. ‘cryptech\_muxd’ does not generate the XDR packaged RPC messages; it only forwards them to the HSM. The RPC messages are generated in ‘libhal’ and ‘libhal.py’.

1. Alpha Board Drawing



1. SLIP

SLIP stands for Serial Line Internet Protocol. SLIP is the result of the integration of modern protocols prior to the suite of TCP/IP protocols. It is a simple Internet link protocol conducting neither address nor error controlii. In the CrypTech project, SLIP is used to send RPC data to the HSM.

Protocol Elements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hex value | Dec Value | Oct Value | Abbreviation | Description |
| 0xC0 | 192 | 300 | END | Frame End |
| 0xDB | 219 | 333 | ESC | Frame Escape |
| 0xDC | 220 | 334 | ESC\_END | Transposed Frame End |
| 0xDD | 221 | 335 | ESC\_ESC | Transposed Frame Escape |

SLIP works by

* appending a special “END” byte which distinguishes datagram boundaries in the byte stream
* if the END byte occurs in the data to be sent, the two byte sequence ESC, ESC\_END is sent instead
* if the ESC byte occurs in the data, the two byte sequence ESC, ESC\_ESC is sent
* variants of the protocol may begin, as well as end, packets with END.

https://en.wikipedia.org/wiki/Serial\_Line\_Internet\_Protocol

CrypTech encode/decode (Python)

def slip\_encode(buffer):

"Encode a buffer using SLIP encapsulation."

return SLIP\_END + buffer.replace(SLIP\_ESC, SLIP\_ESC + SLIP\_ESC\_ESC).replace(SLIP\_END, SLIP\_ESC + SLIP\_ESC\_END) + SLIP\_END

def slip\_decode(buffer):

"Decode a SLIP-encapsulated buffer."

return buffer.strip(SLIP\_END).replace(SLIP\_ESC + SLIP\_ESC\_END, SLIP\_END).replace(SLIP\_ESC + SLIP\_ESC\_ESC, SLIP\_ESC)

1. PKCS 11 Functions

PKCS#11 information from <http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html#_Toc416959728>

Cryptoki's functions are organized into the following categories:

* general-purpose functions (4 functions)
* slot and token management functions (9 functions)
* session management functions (8 functions)
* object management functions (9 functions)
* encryption functions (4 functions)
* decryption functions (4 functions)
* message digesting functions (5 functions)
* signing and MACing functions (6 functions)
* functions for verifying signatures and MACs (6 functions)
* dual-purpose cryptographic functions (4 functions)
* key management functions (5 functions)
* random number generation functions (2 functions)
* parallel function management functions (2 functions)

In addition to these functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

The Cryptoki API functions are presented in the following table:

Table 30, Summary of Cryptoki Functions

| Category | Function | Description |
| --- | --- | --- |
| General | C\_Initialize | initializes Cryptoki |
| purpose functions | C\_Finalize | clean up miscellaneous Cryptoki-associated resources |
|  | C\_GetInfo | obtains general information about Cryptoki |
|  | C\_GetFunctionList | obtains entry points of Cryptoki library functions |
| Slot and token | C\_GetSlotList | obtains a list of slots in the system |
| management | C\_GetSlotInfo | obtains information about a particular slot |
| functions | C\_GetTokenInfo | obtains information about a particular token |
|  | C\_WaitForSlotEvent | waits for a slot event (token insertion, removal, etc.) to occur |
|  | C\_GetMechanismList | obtains a list of mechanisms supported by a token |
|  | C\_GetMechanismInfo | obtains information about a particular mechanism |
|  | C\_InitToken | initializes a token |
|  | C\_InitPIN | initializes the normal user’s PIN |
|  | C\_SetPIN | modifies the PIN of the current user |
| Session management functions | C\_OpenSession | opens a connection between an application and a particular token or sets up an application callback for token insertion |
|  | C\_CloseSession | closes a session |
|  | C\_CloseAllSessions | closes all sessions with a token |
|  | C\_GetSessionInfo | obtains information about the session |
|  | C\_GetOperationState | obtains the cryptographic operations state of a session |
|  | C\_SetOperationState | sets the cryptographic operations state of a session |
|  | C\_Login | logs into a token |
|  | C\_Logout | logs out from a token |
| Object | C\_CreateObject | creates an object |
| management | C\_CopyObject | creates a copy of an object |
| functions | C\_DestroyObject | destroys an object |
|  | C\_GetObjectSize | obtains the size of an object in bytes |
|  | C\_GetAttributeValue | obtains an attribute value of an object |
|  | C\_SetAttributeValue | modifies an attribute value of an object |
|  | C\_FindObjectsInit | initializes an object search operation |
|  | C\_FindObjects | continues an object search operation |
|  | C\_FindObjectsFinal | finishes an object search operation |
| Encryption | C\_EncryptInit | initializes an encryption operation |
| functions | C\_Encrypt | encrypts single-part data |
|  | C\_EncryptUpdate | continues a multiple-part encryption operation |
|  | C\_EncryptFinal | finishes a multiple-part encryption operation |
| Decryption | C\_DecryptInit | initializes a decryption operation |
| functions | C\_Decrypt | decrypts single-part encrypted data |
|  | C\_DecryptUpdate | continues a multiple-part decryption operation |
|  | C\_DecryptFinal | finishes a multiple-part decryption operation |
| Message | C\_DigestInit | initializes a message-digesting operation |
| digesting | C\_Digest | digests single-part data |
| functions | C\_DigestUpdate | continues a multiple-part digesting operation |
|  | C\_DigestKey | digests a key |
|  | C\_DigestFinal | finishes a multiple-part digesting operation |
| Signing | C\_SignInit | initializes a signature operation |
| and MACing | C\_Sign | signs single-part data |
| functions | C\_SignUpdate | continues a multiple-part signature operation |
|  | C\_SignFinal | finishes a multiple-part signature operation |
|  | C\_SignRecoverInit | initializes a signature operation, where the data can be recovered from the signature |
|  | C\_SignRecover | signs single-part data, where the data can be recovered from the signature |
| Functions for verifying | C\_VerifyInit | initializes a verification operation |
| signatures | C\_Verify | verifies a signature on single-part data |
| and MACs | C\_VerifyUpdate | continues a multiple-part verification operation |
|  | C\_VerifyFinal | finishes a multiple-part verification operation |
|  | C\_VerifyRecoverInit | initializes a verification operation where the data is recovered from the signature |
|  | C\_VerifyRecover | verifies a signature on single-part data, where the data is recovered from the signature |
| Dual-purpose cryptographic | C\_DigestEncryptUpdate | continues simultaneous multiple-part digesting and encryption operations |
| functions | C\_DecryptDigestUpdate | continues simultaneous multiple-part decryption and digesting operations |
|  | C\_SignEncryptUpdate | continues simultaneous multiple-part signature and encryption operations |
|  | C\_DecryptVerifyUpdate | continues simultaneous multiple-part decryption and verification operations |
| Key | C\_GenerateKey | generates a secret key |
| management | C\_GenerateKeyPair | generates a public-key/private-key pair |
| functions | C\_WrapKey | wraps (encrypts) a key |
|  | C\_UnwrapKey | unwraps (decrypts) a key |
|  | C\_DeriveKey | derives a key from a base key |
| Random number generation | C\_SeedRandom | mixes in additional seed material to the random number generator |
| functions | C\_GenerateRandom | generates random data |
| Parallel function management | C\_GetFunctionStatus | legacy function which always returns CKR\_FUNCTION\_NOT\_PARALLEL |
| functions | C\_CancelFunction | legacy function which always returns CKR\_FUNCTION\_NOT\_PARALLEL |
| Callback function |  | application-supplied function to process notifications from Cryptoki |

Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes either its entire goal, or nothing at all.

* If a Cryptoki function executes successfully, it returns the value CKR\_OK.
* If a Cryptoki function does not execute successfully, it returns some value other than CKR\_OK, and the token is in the same state as it was in prior to the function call.  If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.
* In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR\_GENERAL\_ERROR.  When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function MUST have a “stub” in the library which simply returns the value CKR\_FUNCTION\_NOT\_SUPPORTED. The function’s entry in the library’s **CK\_FUNCTION\_LIST** structure (as obtained by **C\_GetFunctionList**) should point to this stub function (see Section 3.6).

1. Tornado

From Tornado User Guide – <http://www.tornadoweb.org/en/stable/guide.html>

Tornado is a Python web framework and asynchronous networking library, originally developed at FriendFeed. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

Tornado can be roughly divided into four major components:

* A web framework (including RequestHandler which is subclassed to create web applications, and various supporting classes).
* Client- and server-side implementions of HTTP (HTTPServer and AsyncHTTPClient).
* An asynchronous networking library including the classes IOLoop and IOStream, which serve as the building blocks for the HTTP components and can also be used to implement other protocols.
* A coroutine library (tornado.gen) which allows asynchronous code to be written in a more straightforward way than chaining callbacks.

The Tornado web framework and HTTP server together offer a full-stack alternative to WSGI. While it is possible to use the Tornado web framework in a WSGI container (WSGIAdapter), or use the Tornado HTTP server as a container for other WSGI frameworks (WSGIContainer), each of these combinations has limitations and to take full advantage of Tornado you will need to use the Tornado’s web framework and HTTP server together.

A minimal “hello world” example looks something like this:

import tornado.ioloop

import tornado.web

class MainHandler(tornado.web.RequestHandler):

def get(self):

self.write(“Hello, world”)

def make\_app():

return tornado.web.Application([

(r”/”, MainHandler),

])

if \_\_name\_\_ == “\_\_main\_\_”:

app = make\_app()

app.listen(8888)

tornado.ioloop.IOLoop.current().start()

1. External Data Representation (XDR)

External Data Representation (XDR) is a standard data serialization format, for uses such as computer network protocols. It allows data to be transferred between different kinds of computer systems. Converting from the local representation to XDR is called encoding. Converting from XDR to the local representation is called decoding. XDR is implemented as a software library of functions which is portable between different operating systems and is also independent of the transport layer.

XDR uses a base unit of 4 bytes, serialized in big-endian order; smaller data types still occupy four bytes each after encoding. Variable-length types such as string and opaque are padded to a total divisible by four bytes. Floating-point numbers are represented in IEEE 754 format. iv

1. References

1. <https://trac.cryptech.is/wiki/AlphaBoardComponents>

1. <https://ccm.net/contents/282-ppp-and-slip-protocols>

1. <https://pythonhosted.org/pyserial/>
2. https://en.wikipedia.org/wiki/External\_Data\_Representation