Automated SFC Byte Buffer Handling

Integration Guide

CryptoServer

utimaco®

Imprint

Copyright 2020	Utimaco IS GmbH
	Germanusstr. 4
	D-52080 Aachen
	Germany
Phone	AMERICAS +1-844-UTIMACO (+1 844-884-6226)
	EMEA +49 800-627-3081
	APAC +81 800-919-1301
Internet	https://support.hsm.utimaco.com/
e-mail	support@utimaco.com
Document Version	1.0.0
Date	06/10/2025
Status	**PUBLISHED**
Document No.	IG-2025-0002

Table of Contents

1 Summary

This document describes tools and associated files that assist in the development of host applications and custom firmware for the Utimaco CryptoServer general-purpose Hardware Security Module. The tools described target the communications buffers between a host application, and the custom firmware.

The text below does *not* assume you have had the SDK training, however the training is recommended. It does assume that you have received the SDK installer; the external documention references refer in part to files that are only available as part of the SDK installation.

1.1 Purpose

In general, communications between a host application and a custom firmware module is over both packet and byte-oriented physical media (Ethernet and PCIe bus), using well-formed byte buffers.

A host application identifies which HSM function a byte buffer should be routed to, by supplying a module's *Module ID* (mID), the specific method's *sub-function code* (SFC), and the byte buffer itself to the SecurityServer API library's `.exec(...)` method.

The SDK documentation and examples demonstrate construction and manipulation of these byte buffers manually. The tools described here are _code generation tools_, designed to generate all the byte buffer construction and manipulation up front, and automatically, based on user input.

This is done through the compilation of a simple `pattern`, as used by `cmds_scanf` (discussed later). Alternately, these patterns can be derived from other input formats (IDL, XML, etc), however the primary tool, "SFCBuffers" ingests cmds_scanf patterns. Additional tooling (for non-standard input formats) are provided to consume the other formats, but generate the necessary pattern and supporting fields into the standardized format, for compilation by SFCBuffers.

The string 'u4v1c32v2u4' is an example pattern, suitable for use by SFCBuffers.

As a pattern is compiled, the tools will generate:

- Host source (C++ and Java) classes (OOP) based on the `pattern`. The OOP classes compute serialized length of, and will serialize and deserialize byte buffers based on the pattern,
- Host source test libraries that demonstrate use of the OOP classes,
- A complete SDK "external SFC" method stub, that uses SDK-standard methodology to parse the incoming byte buffer for use by the method,
- C library code that computes serialized length of, and will serialize and deserialize byte buffers based on the pattern,
- Lua code that serializes and deserializes byte buffers based on the pattern, and
- Additional code artifacts, libraries and how-to documents that assist in development.

2 Introduction

The Utimaco CryptoServer is a general-purpose _Hardware Security Module_ (HSM) provided by Utimaco. An HSM is a PCIe-based computing environment -- CPU, memory, storage etc -- with both physical and logical protections that are designed to provide the computing environment with an awareness of its physical surroundings and state. The circuits are what allow the HSM to protect the keys and other secrets that it stores.

Attempts to defeat the physical protections/anti-tamper circuits results in the complete erasure of any stored secrets, sensitive material, etc.

Logical access to the CryptoServer is supported by host libraries that implement both a proprietary cryptosystem (CXI) and several standards'-based cryptosystems (CSP/CNG, PKCS#11, etc), as well as provide the communications protocols, drivers and so on, which are necessary to reach the device.

The current version of the physical hardware is sold as CryptoServer, *and the supporting software libraries, tools and documentation (on the host and on the CryptoServer) is sold as the* Utimaco SecurityServer.

For information on supported *Application Programming Interfaces* (APIs), please see the relevant documentation in the installation, at <install>\Documentation\Crypto_APIs*.

What these host APIs all have in common is an understanding of current, 'modern' generic cryptographic operations, algorithms, mechanisms, etc. Because cryptographic tools are continually being developed or improved, the capabilities of the cryptosystems as shipped may not include awareness of these future mechanisms, or may not be able to provide these new concepts in cryptography. This is currently the issue with those algorithms that are targeting the Post-Quantum Computing era.

For this reason, Utimaco SecurityServer also provides several *Software Development Kits* (SDKs).

Utimaco use of API and SDK: Software that is developed on the host uses one or more "APIs". Software that is developed to run on the CryptoServer is developed using an "SDK". An *API* is always used by host-side application code, and the *SDK* is always used to develop custom, embedded (HSM) firmware.

2.1 The SDK

An SDK is used for two primary purposes, those being *performance enhancement* and *custom logic*. These can be treated individually, or they can be included in a single module. The end user need not target one or the other.

2.1.1 Performance Enhancement/Cost reduction

Also called custom business logic, these are the steps that might take several individual calls to the CryptoServer. Because the CryptoServer may be many hops away (network topologically speaking), a single call may be "expensive" in terms of time or even cost (cloud data charges).

Instead of making these several calls individually, the SDK user can supply all the various inputs to a single custom firmware SFC call, and receive at the end of processing the final result they are targeting.

2.1.2 Custom Logic

For use where the HSM does not provide an out-of-the-box version of some cryptographic operation, for example the upcoming Post-quantum algorithms, or various (regional) nonstandard hashing or encrypting mechanisms, these can be coded up and loaded onto the HSM, for use by host-side applications.

2.1.3 Variants

2.1.3.1 SDK: C

The C SDK is the primary SDK. The programming language is C, and the programmer has full access to the entire OS, platform, hardware (USB, Clocks, etc) available. If it can be done, the C SDK provides the access to do it.

It can be used for both custom firmware, and for performance enhancement.

2.1.3.2 SDK: CryptoScript

The CryptoScript SDK is based on a variant of the Lua scripting language. Your custom code will run sandboxed; only certain access is granted by the sandbox, and there is no arbitrary access of the outside context.

The primary usecase for CryptoScript is performance enhancement.

2.1.4 SDK: PKCS#11 VDM

The PKCS#11 VDM SDK allows the end user to supply custom mechanisms or algorithms.

The resulting VDM code runs inside a sandbox, and there is no access to any external context. It is strictly one operation per call.

Accordingly, the use case is solely custom firmware.

2.2 Why the SDK

For the Enterprise, the Utimaco CryptoServer SDKs represent the ability to migrate business logic to, accelerate cryptographic operations on, or even to produce entirely new algorithms for the HSM.

Because the SDK-developed "custom firmware modules" run directly within the physical security boundary of the HSM, the protection level around the enterprise's intellectual property is elevated to the same as its cryptographic keys, encrypted data, digital signatures and other sensitive material. In effect, enterprise IP becomes a peer of the security protection envelope.

2.3 Protocol Background and limits

The CryptoScript SDK is based on a variant of the Lua scripting language. Your custom code will run sandboxed; only certain access is granted by the sandbox, and there is no arbitrary access of the outside context.

The primary usecase for CryptoScript is performance enhancement.

Whether they are Utimaco standard modules, or SDK-derived custom firmware modules, each module is identified by a module name and ID (mID).

Module IDs from 0x000 to 0x0FF are reserved for Utimaco use. Module IDs from 0x100 to 0x17F are available for SDK user use.

A module is limited to 256 SFC methods.

The maximum communications buffer length is 256 Kilobytes, including overhead.

If you have to send more than 256 Kilobytes of data to a custom firmware module, you will need to develop a protocol for doing so (either store it, or process it inline and return 'state' that allows you to pick up where you left off).

The file size limit for Se Gen2 is 8Mb. The file size limit for Se, and CSe is 4Mb. The row length limit for the database module is 24Kb. In all these cases, assume "less overhead".

The example module shipped with the full SDK is 'exmp', it's mID is 0x100. The exmp module supplies 15-20 different example 'Sub-function' methods for reference purposes, as well as a c++ host application that exercises them.

Collectively referred to as a module's sub-function codes (SFCs), the SFCs represent the external interface of a custom module.

A module *may* provide up to three different type of methods (described below). For convenience, and depending on the mix of these method types, a module may be referred to as an *external interface* module, or as a *utility* module, however there is nothing stopping a utility module from having an external interface, or an external interface module from providing utility methods to other modules.

A module with an external interface provides mID+SFC access to one or more methods. The host application uses one of the APIs to "exec" the SFC, by providing a correctly structured byte buffer. "Correctly structured" here depends on what the SFC is expecting, and will be probably be different for every SFC.

A utility module provides access to its public methods via a 'shared object' paradigm. A utility module does not necessarily provide an external interface, rather its focus is on providing a *public* interface to its methods. Public interfaces are accessable from other *modules*, but not from a *host application*.

All modules will have an *internal* interface, but this interface is only visible to and used within the compiled module.

An external interface module may *also* provide public methods, indeed it may simply be providing an external interface *to* its own public methods, or even to some *other modules'* public methods! For an example of an external interface module that simply provides access to other modules' public methods, see <install>\SDK\doc\mdl_CXI.pdf.

Every SFC will have what amounts to a unique input or output structure (these are not, however, method *signatures* as understood at the function or method level.) You might refer to these as *interface* patterns.

The Utimaco host libraries pass byte-buffers between the host application and the mID+SFC external method. This can be demonstrated using the csadm '`cmd`' command:

'csadm cmd=0x83,1,'abc''

The above csadm command will send the byte buffer `+[0x61 || 0x62 || 0x63]+` to module ID 0x083 (module 'cmds'), sub-function 1 ('reverse echo').

Example csadm 'cmd' call

```
> csadm cmd=0x83,1,'abc'

ANSW:

0 666564 |cba |

>
```

At the enterprise level, the technical term for this type of behavior is Remote Procedure Call ('RPC') and this is generally provided using some form of middleware. RPC is generally implemented for parallel or distributed applications. Additionally, it is usually found in the medium or large scale, enterprise-level application space. RPC middleware packages also tend to add a lot of processing and bandwidth requirements, as they will provide for different quality of service, different platform support, etc. They are traditionally designed for support in "big iron" environments.

Because of the bandwidth and processing requirements, or 'footprint' of an RPC, Utimaco does not support any of the standard tools or middleware for RPC (CORBA, SOAP, RESTful, OMG-DDS, etc), as they add too much of a bandwidth requirement, or the HSMs embedded environment fails to provide the necessary prerequisites. A full-blown RPC also is not generally targeting embedded systems and their limited throughput. This, more than anything, is probably the reason that communication with modules on the HSMs relies on the simplest byte-buffer command/response protocol.

Utimaco provides low-level libraries that handle the passing of secure data between the host and the HSM, but these libraries are only interested in the data at a protocol (secure messaging) and physical (PCIe, Ethernet) levels. The libraries themselves treat the data being passed as opaque.

This means that your host application is responsible for format, and formatting, of the underlying data being passed.

When you target a Utimaco-provided SFC, within a Utimaco-provided module, using a Utimaco-provided cryptosystem implementation (ie, like CXI or PKCS#11), the libraries _do_ provide a higher level API method. These higher-level methods are called with simple parameters, and the method will reformat those parameters into the specific, byte-oriented stream that the target SFC will be able to consume.

In the APIs, the requirement for a "correctly structured byte stream" is abstracted away from the host application.

When writing custom firmware, no such API exists, and must be designed and written at the same time as the firmware itself. The supplied DLLs provide methods to send and receive the byte buffers, but are not aware of how the buffers are structured, or what they contain.

2.4 Sub-Function Payloads for Custom Firmware

Each external interface method (SFC) that a custom firmware module provides may have its own input structure, and may also have its own output structure.

The low-level Utimaco libraries (host and HSM) do provide macros and methods that assist in the correct read and write of a byte buffer, but this assumes that the programmer understands what needs to be *in* the buffer, and what *order* those sub fields need to be in, and, importantly, *how* they should be entered into the byte stream (endianness, length-value protocol used, etc).

For the low-level call to the HSM from host-side code, Utimaco supplies library calls (ie, 'cxi.exec()' or 'cs_exec_command()') which send a previously serialized buffer, along with the routing information needed to target a specific module and sub-function code. Memory management is straightforward: The host application is responsible for both allocation and deallocation of space for the byte-buffer. The various exec methods simply copy from that space (or from a received socket receive buffer) into the PCIe driver's memory-mapped IO buffer.

On the HSM, the module's targeted sub-function will probably use the 'cmds_scanf' methodology (_install_\SDK\doc\mdl_CMDS.pdf) on receipt, to parse the input.

When the HSM sends an answer, HSM to host, the SFC implementation must allocate a buffer for the response within the _parent_ context, eg. by using the Utimaco CMDS-provided 'cmds_alloc_answ()'.

`cmds_alloc_answ()` is part of the CMDS *public interface*.

After the sub-function is finished with its processing, it copies its response into the provided buffer, and returns control to the context. The CMDS context is responsible for transmitting the bytes in the answer buffer. It will then free the memory allocated by the SFC's call to 'cmds_alloc_answ()'.

On receipt, the application may use the 'cs_scanp' methodology to parse the input. This helper routine is described in _install_ \SDK \doc \CryptoServer_CSXAPI.pdf).

Again, all of these input or output byte-buffers are manually serialized by the programmer.

3 Overview of the Firmware Module

Traditionally, a CryptoServer firmware module will have three different types of usable methods: *External*, *Public* and *Internal*.

External methods (names marked with `\*ext*` by convention) are those referred to as subfunction codes in the example module or in the developer documentation. These are the methods that can be addressed directly from a host application, via `cs_exec_command()` or the `cxi::exec()` command, or similar (depending on API).

Public methods (marked by `\*pub*` by convention) are accessible by other modules within the runtime context. When a module wants to access the public API of a different module, it will be provided with the address of the target module's public methods at boot time. Because these are called via the compiler's defined calling conventions (accessed via a function pointer), passing data to/from them is handled via the call stack. For this reason, serialization / deserialization of data is usually not necessary (or is handled by the calling method, before calling a Public function).

Internal methods (unmarked, or marked by `\*int*`) are only visible to the target module itself. These are linked at compile time, and no external symbol is provided in the object.

Again, ordinarily, on the HSM it is only the `\*ext*` methods that need to worry about deserialization on the CryptoServer, or serialization of data for return to the host application.

To assist in deserialization, the `cmds` module's public API includes `cmds_scanf()`, which is a limited-copy parser of structured byte-buffer data. Aside from macros and low-level methods used to append data to a byte buffer, no support is provided for buffer serialization

(on the host or on the CryptoServer). The developer is expected to konw the order, endinanness and protocols for packing the buffer.

3.1 Initial work

When setting up the command interface for use by an `\_ext_` method, the designer will probably start with an idea of what information they need, at a high level. This might be some flags or behavior selection values (passed as an unsigned int with known values), or a hash value, passed as a fixed length byte buffer, or data for encryption or decryption, and/or one or more keys or key handles -- each of which is usually data with no common size from one call to the next.

These data 'fields' will be arranged within a C struct using a format suitable for parsing by `cmds_scanf()`. The `cmds_scanf()` behavior expects unsigned integers, fixed-length and variable-length fields, bit fields, etc.

Here is a complicated input structure, that describes serialized input that contains a flag, a CXI key (by reference or as a blob), a hash value, and some additional data:

A complicated example C Struct

```
struct { // Command struct for sfc 0
unsigned int flags; // u1
unsigned int mech; // u4
unsigned char * keyName; // c32
unsigned char * keyGroup; // c32
unsigned int spec; // u4
unsigned int l_blob; // v2 (length)
unsigned char * p_blob; // (data)
unsigned int l_hash; // v1 (length)
unsigned char * p_hash; // (data)
unsigned int l_data; // * (length)
unsigned char * p_data; // (data)
} cmd;
```

The above struct is a programming aid, helpful in parsing what will be received, and processed. When the host application has the above data, it is in whatever form is normal for the host platform. It might be stack variables, a C struct, or a Java class, but on receipt, the CryptoServer implementation is not passed the above data as maintained on the host, rather it is sent as a stream of raw bytes, after serialization by the host application.

There is no standardized method for doing this serialization, however. Macros are provided that will write values to the serialization buffer (the various load_int/store_int macros in each of the

APIs). These macros are shortcuts, not automation, and are provided mainly to relieve the necessity of worrying about endianness when building the modules and targeting different platforms (the simulator, vs the hardware). Byte buffer copying is done using the OS- or Language-native methods for copying arrays of bytes, possibly with a prepended length value inserted into the byte stream as a single byte, a short or an integer.

The information available when an SFC is triggered is a char * (p_cmd), which points at the raw bytes of the serialized data, and an unsigned integer (l_cmd), which gives the number of bytes pointed to by p_cmd.

The next section describes `cmds_scanf()`, and how it is used to populate the `cmd` struct with information from, or about, the bytes pointed to by p_cmd.

3.2 External Interface Methods ("Sub-Function Codes")

Every external interface method uses the same signature. This signature is defined by Utimaco, and is the signature that the `cmds` module will expect as it calls through a function pointer to the sub-function.

The external-method method signature is a symbol name that takes three parameters: a pointer to the current context, as well as parameters for the length-of-the-payload, and for the payload itself. The return value is an integer, either 0x0 for no-error, or non-0x0 to indicate an error was found.

On the CryptoServer, certain sub-functions take no data, or the single parameter takes up the entire p_cmd field. When the serialized p_cmd data is more complex, a target structure may be provided for use by `cmds_scanf`, and (if used) is generally implemented as a C struct, by convention named `cmd` (as in the example provided above).

int exmp_ext_echo (T_CMDS_HANDLE *p_hdl, int l_cmd, unsigned char *p_cmd);

Because structure packing is not strongly defined by the C language, cross-platform hosting environments that may have different processors, operating systems, programming languages, as well as endianness issues and a lack of automation, it is currently necessary to manually serialize the data to send, and deserialize the data on receipt for each host application, on each platform, for each sub-function and its response.

When a complex structure (`cmd`) is sent, it may include different fields which can represent integers, strings (fixed length character arrays), and/or variable-length byte arrays, or can include something that mimics a switch/case over a union of different structs.

The return interface may be a structure that is as simple, or as complicated, as the input struct described above.

Manually serializing and deserializing data, while straightforward, is repetitive and prone to error. Engineers who have been using the tools provided, often, and repeatedly, may not believe that automation for these steps is of any benefit. For engineers new to the process, or who only use it rarely, automation is a suitable time saver.

Of the four times you will marshal the data -- serialization on the host, deserialization on the CryptoServer, and serialization of the response on the CryptoServer for deserialization on the host, two of these must be done manually. The third and fourth, deserialization of the p_cmd buffer, into the `cmd` struct within an SFC, and deserialization of the response buffer into a similar struct on the host, can benefit from using `cmds_scanf()` (or `cs_scanp`).

Use of `cmds_scanf()` is covered below.

Use of `cs_scanp` (For C and C++ host applications only) is similar, see the relevant documentation for the differences.

In general, the automation provided by the pattern processing tools described here bypasses the use of cs_scanp, by giving you direct access to auto-generated object classes, based on the `cmds_scanf` patterns, where needed.

3.3 Example Walk-through

The architect wants to create a single, custom firmware sub-function that will generate an HMAC for some data, based on a specific digest (SHA512 or SHA2-256 or ...), a secret key derived from some input data, and another key which is stored according to CXI. The CXI key may be passed in either by name/group/spec (to look up an internal key), or as an encrypted 'external key blob'.

Additionally, they may also want to pass in a flag value, that determines how the key should be treated.

The requirement to pass in a key is common, but best-practice is to not define at design time what format the key must be in. Possibilities include a CXI normal 'name, group and spec' identifier, or a key blob which may or may not be wrapped, or it may be some third party or

internal format. And, in many cases, the architect may not even know up front, how the application engineer will implement things on the host.

"As a convention" means for this example. You may have other conventions or requirements.

A reasonable description for the key data, then, is:

- A flag (a 1-byte integer), which may encode different options
- A key Name (using a limit of 32 characters as a convention)
- A CXI key Group (using a limit of 32 characters as a convention)
- A key specifier (a 4-byte integer)
- A data "blob" (an unstructured byte buffer of indeterminate length, generally no more than 4k in size)

Different patterns will match the above requirements. Two such are `u1c32c32u4v2` and `u4v1v1u4v4`. For this walk-through, we will use the u1c32... variant. Patterns and their use are described below.

For the use case, then what the architect might need is:

- The CXI Key (`u1c32c32u4v2`)
  - By reference, or
  - By blob
- MAC algorithm flags (`u4`)
  - Used to select specific functionality within the sub-function
- KDF Data (`v1`)
  - No more than 256 bytes, and
- Data (`*`, ie the remainder)
  - The data to process.

The final pattern is then `u1c32c32u4v2u4v1*`.

The size of a single packet to/from the HSM is limited to 256Kb, including any overhead. If your methods will require more input or output data than that, you will need to develop a method to chunk data, via some sort of 'first, continue, last' methodology.

Per standard SDK SFC design methodology, the developer might create the following SFC method stub, given the u1c32... pattern:

HMAC/CMAC Sub-Function

```
int tcfa_ext_mymac
(T_CMDS_HANDLE *p_hdl, int l_cmd, unsigned char *p_cmd)
{
int err = 0;
struct { // Command struct for sfc 1
unsigned int flags; // u1
unsigned char * keyName; // c32
unsigned char * keyGroup; // c32
unsigned int spec; // u4
unsigned int mech; // u4
unsigned int l_kdfd; // v1 (length)
unsigned char * p_kdfd; // (data)
unsigned int l_data; // * (length)
unsigned char * p_data; // (data)
} cmd;
if ((err = cmds_scanf(l_cmd, p_cmd, "u1c32c32u4v2u4v1*", sizeof(cmd),
&cmd)) != 0)
goto cleanup;
// ...
```

How cmds_scanf works and what it does is covered below. For the reference documentation, see .\SDK\doc\mdl_CMDS.pdf in the SecurityServer installation directory.

The call to `cmds_scanf` will perform different runtime checks on the input data, on the expected number of fields in the cmd struct (computed from the input pattern), and so on, to ensure that

the byte buffer received is at least reasonably well-formed based on expectations. If the return is without error, then:

- cmd.flags (unsigned integer) value is equal to the first byte of p_cmd,
- cmd.keyName points at p_cmd+1,
- cmd.keyGroup points at p_cmd+1+32,
- cmd.l_blob value is equal to the length of cmd.p_blob (and may be zero)
- cmd.p_blob points at the first relevant byte of the p_blob,
- and so on.

The above description covers the case of serialized data arriving at the SFC.

Without the tooling provided, at this point a (host application) programmer would need to develop methods to serialize their host data in either C++ or Java (for use on the host), and may also want to provide a serialization method in C, which could be used on the HSM also -- as the pattern might also be used to describe data being returned (from this or some other SFC).

The tools described by this document use the same pattern as supplied to cmds_scanf, to provide all the missing pieces: Host code object classes in different languages, and HSM code in C, for marshalling data suitable for use with the pattern, as well as test code and other artifacts.

The tools also do not assume that a pattern is an input or output pattern -- when a pattern is compiled, code is generated that allows the pattern to be used immediately for either sending data to the CryptoServer, or getting data back.

4 Use of cmds_scanf

When the input reaches a certain level of complexity, a sub-function code implementation may want to deserialize a received byte array (p_cmd) into a provided structure (cmd) for use.

In general, the method will use 'cmds_scanf()' to deserialize the data, by passing in the two argument values (p_cmd and l_cmd), a 'pattern' value (a series of tokens, see below), and the address and size of the struct cmd stack variable.

Use of 'cmds_scanf()' is not necessary for simple inputs, however the architect may want to use the methods described below anyway. Accepting the p_cmd buffer, as is, can lead to vulnerabilities (buffer overruns, etc). It is recommended to use cmds_scanf as a check to ensure that the incoming data is correctly structured.

cmds_scanf() method use

```
// Functional example of command data parsing

// PATTERN is assumed to be a valid #define, elsewhere

if ((err = cmds_scanf(l_cmd, p_cmd, PATTERN, sizeof(cmd), &cmd)) != 0)

goto cleanup;
```

The scan process has an awareness of the current offset into the byte data, which token (sub-part of a pattern) is current, what the current offset is into the cmd struct, and how many 4-byte entries in the cmd struct are consumed by the current token.

The scanning is also aware of different error conditions that may result from things like the cmd struct size not being correct for the pattern supplied, unexpected end of serialized data, unexpected data remaining at the end of the pattern, etc.

4.1 cmds_scanf Patterns

Patterns are made up of character-length token pairs, like u4, c27 or v2. There is only one single-character token, the optional '+*+', which may only fall at the end of the pattern, if it appears. Aside from the '+*+', all tokens include both a character indicator, and a numeric size.

The complete list of format specifiers for a valid pattern can be found in the 'cmds_scanf()' documentation in mdl_CMDS.pdf.

Our CXI key pattern: u1c32c32v2
Another pattern: u4c64u1u2v2v2v1
A pattern with a 'remainder' mark: u1c32c32v2*

4.1.1 u - The unsigned int

A sub-pattern of 'uN' (N = 1..4) indicates that the serialized buffer will contain (at the current offset) an unsigned (big-endian) integer across 'N' bytes. The 'u' pattern consumes one field in the cmd struct.

This value will be copied to the working field in the target structure. All 'u' pattern values are casted to unsigned (4 byte) int. The four 'u' sub-patterns are 'u1' (a single unsigned char), 'u2' (an unsigned, big-endian short), 'u3' (an unsigned big-endian, 3-byte integer), and the described 'u4'. Again, these are promoted to 4-byte integers when copied by value into the target struct.

4.1.2 c - The fixed-width byte field

The sub-pattern cN (N is greater than 0, like 'c27') is used to indicate a fixed-length field of 'N' bytes. A pointer to the first byte in the field will be put into the current field of the cmd struct, and the offset will be advanced by 'N' bytes. The 'c' token consumes one field in the cmd struct.

The assumption is that the programmer knows that this unsigned char * field points to a char buffer of 'N' bytes. No "l_" value is expected in the cmd struct, no preceding length byte or bytes is expected in the p_cmd data, and no l_ value is supplied to assist the programmer. If 'l_' type information is expected or required, use a 'v' token instead.

4.1.3 b, k, ... -

The pattern documentation includes two additional format specifiers, b and k. These are currently not supported in the pattern compile, however adding them would be straightforward as they are variations on the c specifier.

In general, you can manually substitute a cN pattern where a b or k might be used (assuming fixed length field), according to the following notes.

- b (bit streams) are "cN"-like, with N being treated as bits
    - b8 is equivalent to c1, b24 is equivalent to a c3, etc.

        - Substitution:
            - b to c: Divide N by 8.
    - k (byte streams) are "cN"-like, with N being treated as 8 bytes
        - k1 is equivalent to c8, k3 is equivalent to a c24, etc.
        - Substitution:
            - k to c: Multiply N by 8.

4.2 In use

Simple cmds_scanf() v2 example

```
struct {
unsigned int l_data;
unsigned char * p_data;
} cmd;
// 'BC' endoded as a v1: 024243
// 'BC' encoded as a v2: 00024243
// 'BC' encoded as a v3: 0000024243
// 'BC' encoded as a v4: 000000024243
// In all these cases, 'cmds_scanf()' will populate l_data with '00000002',
// and p_data will point at the byte with 0x42 in it.
// Note the big-endian encodings for the length bytes.
```

All integer values (whether data or length values) should be encoded/entered into the serialized data as big-endian values.

Here is a carefully crafted cmd struct, designed to demonstrate what can go wrong:

Simple example of a Problematic C Struct

```
struct { // Command struct for sfc 0

unsigned int flags; // u1

unsigned int mech; // u2

unsigned int spec; // u4

} cmd;
```

A *PATTERN* for this structure (as commented) is *u1u2u4*. An equally correct pattern is *u2u1u4*. Or *u4u2u1*. Each of these patterns have the correct number of tokens, and the input buffer size will match what the tokens tell cmds_scanf to expect.

To the cmds_scanf command, these three patterns ***match the cmd struct given***. What will differ is how the incoming buffer was serialized on the host, how p_cmd is interpreted, and **what the effective result values in the cmd struct are**. The call may return no error -- but the populated struct would have incorrect information in it.

The cmds_scanf method is smart enough to identify when the p_cmd length differs from what the pattern tells cmds_scanf to expect. In the above example, if data is supplied for the terminal token ('v1' or '*') the pattern is accepted. If no data is supplied (valid when '*'), the 'v1' pattern will generate a deserialization error (unexpected end of input).

If the host programmer is misinformed or doesn't understand how things are parsed on the CryptoServer, they could easily provide the correct data, but be responsible for an incorrect serialization of that data. An incorrect serialization would result in either the parse failing, or worse, the parse succeeding but the data being corrupted.

Common failure modes:

- Adding a length value into the data when the pattern expects a 'remainder' mark, '+*+',
- Using an incorrect length mark size (serialized v4, expected v2),
- Two integers like `u4u4`, but on the host they are encoded as "flags, then specifier", but on the HSM as "specifier, then flags".

- ...

The cmds_scanf method is a "limited copy" parser. It is limited copy, because integers (the length fields of 'v' tokens, or 'u' tokens) are copied by value, but data fields (for the 'v' and '*' tokens) are pointed to.

Do not attempt to modify data pointed to by a field in the cmd struct, as populated by cmds_scanf -- the sub-function code does not 'own' that data.

Because data fields are pointers, subsequent code can use the struct members for read-only access to data within the passed-in p_cmd data. However: If the code tries to modify the pointed to memory, results are indeterminate (the p_cmd data is 'owned' by the parent context, not by the sub-function method).

Again, the programmer is expected to understand how the pattern scanning works on the HSM, what is required in the interface (based on each field within the pattern), the order of the fields being serialized, and also is expected to know what is best practice on the host, for assembly of the serialized data in a given language, taking into account endianness, length markers for 'v'-type fields, etc. The programmer is also expected to understand that the CryptoServer is big-endian, that a command byte array (or response) is limited to 256Kb of data (less some overhead), etc.

There are, in short, several different failure modes that automation could check for, or ensure do not happen.

While cmds_scanf-like methodology is provided on the host, it is not available in the high-level (cryptosystem) APIs, only in the low-level libcsx \(C) api.

5 Automation

The purpose of the tools provided is to allow the programmer (or, more likely, the system architect) to auto-generate much of the code necessary for handling the communications pathways, between a host application and the CryptoServer module.

The architect starts with an idea of what data needs to be transmitted, and in what order. From that, a pattern can be derived.

The tools will "compile" the pattern, and from that generate artifacts and code based on supplied templates, for both the HSM custom firmware module, *and* the host. This includes things like the C cmd struct (for the SFC '*ext*' methods), additional boiler-plate code around the C cmd struct, artifacts that provide best-practice serialization/deserialization, test code and example usage, in different programming languages and for different platforms.

It is assumed that certain of the output from the tooling will need to be merged into other files. An eventual goal is to provide an input configuration file that defines all patterns, symbol names, sub-function method information for an entire module, and therefrom auto-generate the complete structure (as if this were UML-defined). Indeed, this would allow the use of UMLbased tooling to provide additional levels of control and code generation.

For the below discussion, access to the Utimaco CryptoServer SDK is assumed. It will be useful to have access to the SDK/doc/mdl_CMDS.pdf (where the documentation for cmds_scanf is found, where the pattern structure is defined), and the example SDK/cs2/mdl/ exmp module and its src directory, to compare the output of the tools to what is found in that module in the SDK.

5.1 Tools

5.1.1 SFCBuffers.pl

SFCBuffers.pl is the front-end to the complete tool set.

SFCBuffers.pl Use

```
[perl] SFCBuffers.pl -h

Usage:  [perl] SFCBuffers.pl [-h]

        Display this help text.

Usage:  [perl] SFCBuffers.pl [-pattern] pattern <args...>

        The -pattern flag is optional. If not seen, all command line args

        will be scanned, looking for the first parameter that matches a

        cmds_scanf-format string.

        Optional args and default values/notes:

        -cdb <file.cdb> Will parse the file.cdb for arguments/flags to use,

        rather than assume they are found on the command line.

        -cname <CommandName> : cmd

        [-pattern] <pattern> : u1u2u3u4c15v1v2v3v4k2*

        -fields <name,name,name,...> :

        aByte,aShort,aTripl,u4ea,chuck,veeOne,veeTwo,veeTree,veeFour,\

        mountain,bunker

        -class <ClassName> : GenericSFC (struct {...} GenericSFC;)

        -dev <devstring> : 3001@127.0.0.1

        -go : (accept all input, no interaction)

        -mdl <ModuleName> : exmp (cs2/mdl/<ModuleName>/...)

        -mdl_id <module_id> : 0x100 (#define MODULE_ID 0x<mdl_id>)

        -sfc <sfc_id> : 1

        -files <glob> : *.*

        -plist <Languages> : C:Cpp:Java:Txt
```

```
        -t <templateDirPath> : ./templates/

        -answ <AnswStruct> : (unset by default)

        -v  : (verbose)

            : If verbose is set, the script will dump a .cdb file format

            : that provides this configuration.

        -get <prefix:suffix> : get: (get<FieldName>, for each FieldName)

        -set <prefix:suffix> : set: (set<FieldName>, for each FieldName)

        -D<flag<=var>> -- user-defined flags
```

- -cdb <file.cdb>
  The tools accept a 'colon-delimited database' file, similar in use to most config files. The format of a row in the .cdb file is

  ```
  ::TAG::fob data[:...]
  ```

  If there are no colons in the data field, it is treated as a scalar. If there is/are, then the data field will be split on the : and treated as an array. Because arrays require internal handling on a case by case basis, they are limited to ::IN:: type tags (see below). Specifically, other type tags (CDB, FLAG, etc) are not array-aware.

There are several different example .cdb files supplied in the cdb directory in the installation.

In general, a .cdb file is a superset of the command line arguments available -- Certain type tags found in the .cdb files have no analogue with the command line arguments. To generate a .cdb file based on the current input arguments, which can then be edited using any common plain-text editor, use the -v flag on the command line. The .cdb structure that corresponds to the other input command line arguments will be output to the command line; exceptions will be noted.

- ::IN:: tags in a .cdb file are treated like command line arguments; they generally have side effects (additional processing happens based on these settings).
- ::FLAG:: tags in a .cdb file are local arguments, and do not have side effects in the basic installation. It is possible to extend the scripts to treat specific ::FLAG:: tags with additional processing. These extensions are local installation specific. Additionally, later revisions of the installer will not place anything in the relevant directories (see the sub-directories under Personalities for examples).
- **-cstruct <StructName>** {nbsp}{nbsp} *default: cmd*
  - CDB: **::IN::StructName cmd**
    By default/per convention, the output sub-function code stub will use 'cmd' as the generated c struct name for use by cmds_scanf. To use a different name, set it here and it will be propagated where necessary (for example, the generated cmds_scanf call will reference the new CommandName, rather than cmd).
    This will not change the behavior of the internal generated code; the "response" struct will still be the lower-cased classname value.
- **[-pattern] <pattern>** {nbsp}{nbsp} *no default*
  - CDB: '::IN::inpattern <pattern>'
    The -pattern mark is optional. If not found, all arguments will be parsed looking for one that matches the cmds_scanf pattern semantics.
- **-fields <fieldname:fieldname:fieldname:...>** {nbsp}{nbsp} *no default*
  - CDB: **::IN::innames <name,name,name,...>**
  - CDB: **::IN::fieldtags <lenSpecifier>:<ptrSpecifier>[:<postfixBool>]**
    These are the names of the fields (one per token) that match the supplied pattern. The array of names are separated on the command line or in the .cdb file using a comma. Use of a : will cause them to be split into an array too soon, so as a special handling step, if you accidently use : here, it will be converted back to commas before processing.
    Pattern marks that result in two cmd fields (vN, *, etc) still only require a single name. vN field names will prepend either l_ or p_ (or see below) to match the multiple fields requirement in the structure.
    The initial l_ and p_ for multi-field values can be changed via .cdb file input as shown above. Whatever is in front of the : will be placed in front of the provided name for length fields, and whatever is after will be placed in front of the provided name for pointer fields. At least one must be provided ('l_:' and ':p_' are valid, and will result in 'l_field'/'field' and 'field'/'p_field' respectively). The postfixBool field may be 0 (default) or 1. When 1, the values supplied for the specifiers will be postfix modifications instead of prefix.

On processing, fieldnames (like 'flags,key,cruft') will be matched to sub-pattern specifiers (like 'u4v2*'), from left to right. It is expected that the sub-pattern count should equal the fieldname count, but it is not an error if there is a mismatch. The script will match sub-patterns to fieldnames so long as they are available, and then request additional (if necessary) at the command line. If the pattern requires fewer fieldnames than supplied, a *warning* will be shown, and remaining fieldnames will be discarded.
There is a special case for the '+*+' (remainder) mark, in that if a fieldname is not supplied for it, "bkkt" (bucket) is used.

cf comments in *-go*, related to warnings. In this case, if the fieldnames supplied don't match to the pattern specifiers, the tool will exit with an error message.

The postfixBool flag is not currently supported.

It is possible to use sub-structure patterns, like 'u4u4<key>v2', and a matching 'flag,mech,key,cruft' name list. This is specific to this tool. Please see the section below that describes that usage.

- **-class <ClassName>** {nbsp}{nbsp} *no default*
  - CDB: **::IN::ClassName <ClassName>**
    This is the class name used for the host-side OOP languages (Java, C++). Additionally, it is used as the struct name when this pattern is used as a ***response*** value.
    When C is generated, you will get both a cmd struct in the generated _ext_ stub, and a <classname> struct (note the de-capitalization) in the <ClassName>.h header file:

```
// In <ClassName>.h

struct {

  unsigned int    l_data; // v2

  unsigned int    p_data; //

} classname; // but notice the capitalization
```

```
// Directly in the _ext_ stub code:

struct {

  unsigned int    l_data; // v2

  unsigned int    p_data; //

} cmd; // but see also -cname above
```

- * **-dev <devstring>** {nbsp}{nbsp} *default: 3001@127.0.0.1*
  - CDB: **::IN::dev <devstring>**
    The supplied device string is used for the generated Test applications, and is in use the same as ***Dev=<devstring>***, as used by csadm, cxitool, etc.
- **-go**
  - CDB: **::IN::go [1|0]**
    If you don't need interactivity, -go will cause the tools to run without.
    Things which are warnings while interactive are promoted to errors when in -go, and the tools will not run to completion.
- **-mdl <ModuleName>** {nbsp}{nbsp} *default: exmp*
  - CDB: **::IN::ModuleName <ModuleName>**
    Used to tell the tools what the target module name is (SDK/cs2/mdl/<modulename>), like 'exmp'
- **-mid <module_id>** {nbsp}{nbsp} *default: 0x100*
  - CDB: **::IN::mID <module_id>**
    Used to tell the tools what the target module ID is (0x100 .. 0x17f). There are some

heuristics to generate a valid mID based on the input. If it fails, you get a warning and the value 0x13a is substituted for the unexpected module id value.

cf behavior for -go

- Single SFC:
  - **-sfc <sfc_id>** {nbsp}{nbsp} *default: 0*
  - CDB: **::IN::SFC <N>**
    Used to tell the tools what the target sub-function code for this pattern is.
- A complete Module's SFCs
  - Repeatable: **-sfc <sfc_id>:<sfc_name>:<inpattern>[:<outpattern>]**
  - **::IN::SFC_Count <N>**
  - **::SFC::0 <name>:<AStructName>[:BStructName]**
  - **::SFC::(N-1) <name>:<AStructName>[:CStructName]**
  - ::STRUCT::AStructName <pattern>:<fieldname>[:<fieldname>[:...]]
  - ...

The IN::SFC_Count value is a hint to the loop of how far to look for different SFC::fob indexes, 0..(N-1) will be looked for. Values can be skipped. For each found, the template processing will generate an _ext_ stub method, with the input and output structures and handling based on the STRUCT::<StructName> values requested (behavior for the given struct will be determined by its location in the ::SFC::n line).

If the SFC does not process the input l_cmd/p_cmd data, or expects a zero-length p_cmd value, the input struct name can be absent:

```
`::SFC::5 getdefaultkey::KeyBlob`
```

Module SFC process is not yet supported.

- **-files <glob>** {nbsp}{nbsp} *default: *.**
  - CDB: **::IN::files <glob>**
    By default, the template processing engine will attempt to process all files found in the templates directory, based on patterns built into the file names themselves. The -files argument can subset this, to only target a specific file or set of files. The actual

value supplied will be treated as if it were wrapped by wildcards, so a '-files EXT' will cause processing to happen to any file with EXT in the filename.

To limit template processing to a specific personality (or personalities), see *-plist <Languages>* . -plist and -files can be used together.

- **-plist <Languages>** {nbsp}{nbsp} *default: C:Cpp:Java:Txt*
  CDB: '::IN::plist C:Cpp:Java:Txt'

  Ordinarily, the template processing engine will attempt to process all files found in the templates directory, based on file type. Each supported personality (see the .../Personalities directory) will have its own sub-directory in the templates directory, which provides the files it expects to work on.
  To limit processing to one or more specific personalities, use plist to list it/them.
  To process specific files based on filename, see *-files <pattern>* . -plist and -files can be used together.

The separator for the fields supports either `,` or `:`, when used in the .cdb file.

- **-t <templateDirPath>** {nbsp}{nbsp} *default: ./templates/*
  - CDB: **::IN::template <templateDirPath>**
    The root directory of where the templates should be sourced from.
- **-answ <SymbolName>** {nbsp}{nbsp} *default: (unset by default)*
  - CDB: **::IN::AnswStruct <SymbolName>**
    If set, the value supplied will be substituted into the generated artifacts where an Answer Struct is used or expected. The templates use the replacement tag #AnswStruct#. Also, The 'AnswStruct' replacement tag symbol name is treated both as the symbol to use, but also its existence is seen by the preprocessor as the token to cause the output to be there:

```
::IFDEF:: AnswStruct

// information related to the use of an AnswStruct (-answ FooStruct, for example)

// ...

::ELSE::

// no answer struct symbol name seen (ie, no -answ <...> on the command line).

::ENDIF::
```

- -v {nbsp}{nbsp} *default: (unset by default)*
  Slightly more verbose output.
  Also, If verbose is set, the script will dump a .cdb file format that provides the command-line supplied configuration. The format, saved as a .cdb file, can be used via -cdb <format> to rerun the same configuration. CDB files can be edited.
- -get <prefix:suffix> {nbsp}{nbsp} *default: 'get:' (result is get<FieldName>, for each FieldName)*
  - CDB: ::IN::get <prefix>:<suffix>
- -set <prefix:suffix> {nbsp}{nbsp} *default: 'set:' (result is set<FieldName>, for each FieldName)*
  - CDB: ::IN::set <prefix>:<suffix>
    These provide support for different coding conventions, for getters and setters.
    The get and set targets expect value pairs (the two values are separated by a colon, which must be there).
    The field name will be capitalized for this use.The default values, applied to a symbol name foo, result in getFoo(...) and setFoo(...). If you move the get/set to after the colon, the result would be Fooget and Fooset. You can put something on both sides, like public:Set, and the result would be publicFooSet(...).

5.2 User Defined Flags

- -D<flag[=value]>
  - CDB: ::FLAG::<flag> <value>
    User defined flags can be set. These are treated by the templatization engine, but

they are ignored by the processing script, or the preprocessor. Specifically, -DAnswStruct=foo will *not* be seen by the preprocessor:

```
::IFDEF:: AnswStruct

// information related to the use of an AnswStruct

// ...

::ELSE::

// no answer struct seen, or it was a -DAnswStruct=value

::ENDIF::
```

FLAG values are also not known by the processing script, so are in general less likely to have side effects related to their use in templates.

FLAG values are limited to scalar values (no arrays). If you want FLAG arrays, use something like

```
::FLAG::tag1 value1

::FLAG::tag2 value2

...
```

and template replacement tags like #tag1# and #tag2#.

If you want to set a replacement tag like #Foo# to "" (the empty string), -DFoo is sufficient.

5.3 Special Handling Cases

5.3.1 CMDS Pattern sub-patterns

The inpattern supplied for the main Class can include named sub-patterns. The format in the .cdb requires two CDB entries, one with just the name, and one with the name, plus 'names':

```
::CDB::somekey u4u2...

::CDB::somekeynames flags,ashort,...
```

The number of tokens in the sub-pattern must match the number of names supplied for it.

On ingestion via the CDB processing, when a <somekey> token is seen in the main inpattern, the <somekey> in the main pattern will be directly replaced by the somekey sub-pattern. The list of innames for the inpattern should contain a token, which will be inserted into the sub-pattern names as a mark. Given

```
::CDB::mykeytype u4v1v2u4

::CDB::mykeytypenames flag,name,group,spec

::IN::inpattern u1<mykeytype>v2*

::IN::innames mech,key,data,cruft
```

The end result is the same, as for following pattern being processed:

```
::IN::inpattern u1u4v1v2u4v2*

::IN::innames mech,key_flag,key_name,key_group,key_spec,data,cruft
```

The use case for this is the ability to use sub-patterns that pop up often (key name/group/spec for example), and also the ability to +duplicate+ sub-patterns in the same main pattern:

```
::IN::inpattern u1<key><key>v2*

::IN::innames flag,inkey,outkey,data,cgram
```

5.4 Templates

The tools are built on a template processor built in perl.

By default, the templates directory is the ./templates directory in the installation. This can be overridden at the command line ('SFCBuffers.pl ... -t MyTemplates').

The templates directory has sub-directories with the names of the 'Personalities' that are expected to work on the files within. The C.pm personality expects a C sub-directory, etc.

In general, it makes more sense to add a different Personality (and its templates) to the existing infrastructure, unless you are entirely redoing the output for everything.

5.5 Preprocessor

The preprocessor supports:

5.5.1 Conditional Compilation

- To check the value of a symbol against a static value, equality or inequality:
  - ::IF:: <symbol> == <value>
  - ::IF:: <symbol> != <value>
- To check for the existence of a symbol, regardless of value:
  - ::IFDEF:: <symbol>
- To check for the non-existence of a symbol
  - ::IFNDEF:: <symbol>
- To choose the other boolean path after the above:
  - ::ELSE::
- To terminate an IF, IFDEF, IFNDEF or ELSE block:
  - ::ENDIF::

Each IF, IFDEF or IFNDEF block must be terminated by an ENDIF tag.

Each IF, IFDEF or IFNDEF *may* include an ELSE block, prior to the ENDIF tag.

IF, IFDEF and IFNDEF blocks *may* nest.

The template preprocessor uses a non-standard engine so as to not interfere with the C personality templates' use of the gnu cpp preprocessor directives.

5.5.2 Preprocessor Symbols

A template-specific preprocessor symbol is defined in a template file like:

```
'::SYMBOL::tag [<optional value>]'
```

Note that the SYMBOL (the name) will be lower-cased for use. The above would be seen as \#symbol:tag# in a template where used.

The preprocessor symbol values can themselves contain \#<symbol># constructions, allowing you to use dynamic values inside the file-only values, to target external behavior (Changing the filename, as an example usecase).

Also, preprocessing can include other files, and these other files may provide "template-specific" variables, even though they are not defined directly in the template.

No attempt is made to find circular references. If \#<symA># refers to \#<symB>#, and the inclusion path eventually tries to include \#<symA># again, the result is undefined. A deep-recursion, or out-of-memory crash are the most likely results.

5.6 Templatization

5.6.1 Non-standard input

The symbol table generated by the script, and handed to the language personalities, can have any sort of replacement tags and associated values. The structure is a top level table (generated by IN-based values), which includes a FLAG table and a preprocessor generated symbol table. The order of precedence is Preprocessor > FLAG > IN. In addition to the standard values created for an IN-based input, the standard value may also generate sister values, generally based on different capitalization rules, or on handling of arrays. For example, the CommandName value "FooFnorb" will result in commandname "foofnorb", and COMMANDNAME "FOOFNORB" respectively. For array behavior, see the comments above on the -set and -get command line arguments.

FLAG symbols have no additional processing, they are simply 1:1 replacements.

Arbitrary, additional templates may be supplied for own use. These may require their own replacements, which can be supplied as the normal IN-type symbol replacements, or FLAGbased entries in a .cdb file or preprocessor directives (for static replacements), or they can be the target

for a dynamic replacement, if a Personality is created or extended to be aware of them (see below).

The decision to base additional inputs on IN, FLAG, preprocessor or custom depends on if additional processing is expected on the input data (which is the case for handling arrays). In general, IN-based values are added when extending or developing new personalities or when extending the outer scripts or utilities. FLAG-based values are much less intensive and can be added quickly to existing templates without much effort, obviating the need for custom tags.

5.6.2 References in Templates

Referenced environment variables in templates are marked by \#<variable>#, with the following considerations:

- IN-based variables may exist in differently capitalized variations, foofnorb, FOOFNORB, fooFnorb, etc, and so may be seen as \#foofnorb#, \#FOOFNORB#, etc.,
- FLAG-based variables ignore the 'flag', and are still referenced as just the variable name. A ::FLAG::fob value would be reference using simply \#fob#, and
- PreProcessor-based variables are lower-cased (the name) but are case-dependent for the fob. The name and fob are separated by two colons. A template file that defines ::MYPREP::fooFnorb would use \#myprep::fooFnorb# to reference it in that template.
- Custom flags defined in the .cdb file, for example `::MYCSTM::bar torque` would result in replacements of \#MYCSTM::bar# with `torque`.

5.7 Personalities

Supplied with the distribution are five example personalities, for the C language, the C++ language, the Java language, the Lua (CryptoScript) language, and a generic Text processing variant.

The different personalities are set up to include chunks of additional processing modules, SubPersonalities. An example SubPersonality operation is included at Personalities\C\KeyRef.pm . Use of SubPersonalities requires template changes in the parent Personality templates. If these templates should be seen as unmodifiable, consider creating a new Personality instead, and adding its respective template sub-directory.

5.8 Test files

Various template sets include Test libraries. If a test file is not needed for a given input (.cdb) file, set ::IN::notest in the .cdb file.

6 What is Gained

What is gained is a much faster SFC implementation, and much less error-prone data handling.

When the code is auto-generated, serialization, deserialization and such is handled "correctly" (ie, in a method that is "best practice", based on the language/platform targeted by the artifacts that are output for them).

Reformatting a pattern is as simple as changing the input and recompiling. The output can be merged with an existing method on the CryptoServer, or it can be used as a new SFC. Java or C++ host applications can use the regenerated OOP classes directly, and a decent GUI programming IDE would assist in updating existing applications to use the new classes. There is less chance that code artifacts are missed (resulting in a disconnect between serialized buffers and deserialization attempts).

Time-to-market is significantly improved.

When an implementation of the target functionality already exists in C, it can rapidly be converted to a *public* method, and then wrapped with an *external* method, simply by conversion of its method signature to a cmd struct, matching
that struct to a pattern, and recompiling the module.

6.1 Advanced usage

The techniques described in this section are not yet considered stable. They are not included in this version. A version that allows for -experimental use may be available from Utimaco. Please contact support-cs@utimaco.com for up-to-date information.

6.1.1 Sequences of Structs

The generated OOP artifacts can be generated in such a way to take advantage of parent/child class objects.

Take for example a "KeyID" identifier pattern, made up of two c32 (fixed length fields) and two u4 (unsigned int).

- KeyID: u4c32c32u4 (Flags, KeyName, KeyGroup, KeySpec)
- A KeyID.java class has four members,
    - int flags;

- byte [] KeyName = new byte[32];
- byte [] KeyGroup = new byte[32];
- int spec;

With the above, it becomes possible to generate structs of structs, ie with patterns like `u1v2` where the u1 encodes a '3' for example, and the v2 encodes three separate, serialized, KeyID structs.

Longer term, it should be possible to provide a pattern methodology that accepts pattern identifiers within the pattern like `u2s2u4*:KeyID`. The compiler would then generate the `u2v2u4*` suitable for the module's top-level sub-function `cmd` struct, and then include the necessary "stacked" cmds_scanf() calls to first parse `cmd`, then to parse the `v2` into a keyid struct.

On serialization, a "KeySet" object would first compute the sizes of each KeyID, and then arrive at a final serialization length for itself, including its own requirements and structure. For each KeyID in the array, it would:

1. Capture the serialized length (Added to a running total),
2. Add 4 bytes to the running total,
3. Create a TL buffer with the length of the child object, and then the serialized child object, which it would append to the running byte buffer.

The above results in a "vN" like "unsigned int l_keys; unsigned char * p_keys". The [lp]_keys values could then be passed to common internal functions that treat it as a standardized forward-scannable list:

[source,text]

```
...
struct {
unsigned int cKeyID;
unsigned int cKeyIDBfrLen;
unsigned char *cKeyIDBfr;
unsigned int flags;
unsigned int l_data;
unsigned char *p_data;
} cmd;
struct { // KeyID
unsigned int flags;
unsigned char *KeyName;
unsigned char *KeyGroup;
unsigned int spec;
} keyid;
struct keyid * p_keyid;
// parse command data
if ((err = cmds_scanf(l_cmd, p_cmd, "u1v2u4*", sizeof(cmd), &cmd)) != 0)
goto cleanup;
unsigned LIST * list = // prepare list struct
sdki_list_first(cmd.cKeyIDBfrLen, cmd.cKeyIDBfr);
while (!err && list)) {
struct { // KeyID
```

```
unsigned int flags;
unsigned char keyName[32];
unsigned char keyGroup[32];
unsigned int spec;
} keyid;
// parse command data
if ((err = cmds_scanf(list->length, list->data, "u4c32c32u4",
sizeof(keyid), &keyid)) != 0)
goto cleanup;
if (!err) {
// . . . process the first item
err = sdki_list_next(list); // advances data to point at next field
}
}
...
```

6.1.2 The Semantics of *

The semantics of * can be targeted to allow type patterns that implement C unions.

The cmds_scanf pattern infrastructure understands * to mean "the rest of the data" and has specific behavior to calculate the length of that remaining data. It can only be used as the last sub-pattern within the outer pattern.

The cmd struct pattern can include an integer that acts as a descriminator. This discriminator determines what 'sub' structure is encoded in the * semantics.

Code could be autogenerated to populate the necessary values (discriminator, final field) in the primary cmd data based on values in the host class, and additionally 'stack' the underlying `cmds_scanf()` calls in the external function call.

6.2 Conclusion

In the end, the only thing that the SFC programmer need worry about is the implementation of what they are trying to accomplish -- and not the buffer passing, or data validation for simple things like array sizes, for limitations on the interface or platform realities like endianness.

The achieveable goal is to allow the host application programmer to only worry about Objects -- no need to worry about buffer passing, serialization or data validation.

7 Additional Information

Additional information about the preprocessor behavior, custom templates, etc can be found by browsing through the various examples.

- Templates:
  - `{install}/templates/readme.txt`
- Preprocessor:
  - `{install}/pm/Preprocessor.pm`
- Personalities:
  - `{install}/Personalities/readme.txt`

7.1 Extensibility

Templates and Personalities are extensible.

[.compact]

This is a test.

This is a test.

This is a test.

---

8 Complete output example

The descriptions below take for input the following conditions:

- A pattern, MyPatt, of u1u4c16c16u4v2*
  - By convention, the sub-function method name will be taken from 'MyPatt'
  - Field names of flags, mech, keyName, keyGroup, spec, hash and data
- A module, abbreviated as tCFA with ID of 0x116
- An Answer pattern, AnswPatt
  - Note that the structure of the AnswPatt pattern is (at this point) irrelevant.
  - The AnswPatt might be passed through the automation code as a separate step, but for here it is only needed as a symbol name.
- A package name (used for generated host code), com.utimaco.cs2.mdl.tcfa

8.1 Autogenerated C (CryptoServer SDK module destined code)

Output from compiling the pattern:

1. The representative `struct { ... } cmd;` struct,
2. A boilerplate sub-function _ext method, around the struct, which includes the `cmds_scanf()` call including the pattern,
3. Complete internal function declarations and definitions for serialization, deserialization and length computation of data in the struct, for when used as an answer structure, and
4. Additional internal functions, methods, defines and constants that provide shortcuts on use and prevention of common errors.

Example _ext method implementation stub - no answer expected

```
/*
   - 01a autogenerated 2019-02-25 using
   - SFCBuffers.pl version 01d,20190225,rip
 *
   - For MyPatt using u1u4c16c16u4v2*
 *
   - For merging into an existing tcfa_ext.c look for the ::CUT:: marks.
 */
#ifndef _MyPatt_ext_c_stub_1_0_0_0
#define _MyPatt_ext_c_stub_1_0_0_0
/* Command Struct */
#include <MyPatt.h>
#include <MyPatt_int.c>
//::CUT::
/
*********************************************************************************
***
   - Module:  0x116 (tcfa)
   - tcfa_ext_mypatt
 *
   - Description  :
 *
```

- Command data : SFCBuffers-generated for cmds_scanf
- Pattern: u1u4c16c16u4v2* (MyPatt)

```
 *
 ************************************************************************
****/
int tcfa_ext_mypatt
    (T_CMDS_HANDLE *p_hdl, int l_cmd, unsigned char*p_cmd)
{
  int      err = 0;
  struct { // MyPatt
    unsigned int  flags;                    // u1
    unsigned int  mech;                     // u4
    unsigned char *keyName;                  // c16
    unsigned char *keyGroup;                 // c16
    unsigned int  spec;                     // u4
    unsigned int  l_hash;                   // v2
    unsigned char *p_hash;                   //
    unsigned int  l_data;                   // *
    unsigned char *p_data;                   //
  } cmd;
  // parse command data
  if ((err = cmds_scanf(l_cmd, p_cmd, "u1u4c16c16u4v2*", sizeof(cmd),
&cmd)) != 0)
```

```
        goto cleanup;
  // ... Do Work
cleanup:
  return err;
}
//::__CUT__::
#endif
----
```

```
==== Example _ext method implementation stub, including answer struct
[source,console]
.Generated C Stub (external interface code)
----
/*
  ▪ 01a autogenerated 2019-02-25 using
  ▪ SFCBuffers.pl version 01d,20190225,rip
 *
  ▪ For MyPatt using u1u4c16c16u4v2*
 *
  ▪ For merging into an existing tcfa_ext.c look for the ::CUT:: marks.
 */
```

```
#ifndef _MyPatt_ext_c_stub_1_0_0_0
#define _MyPatt_ext_c_stub_1_0_0_0
/* Command Struct */
#include <MyPatt.h>
#include <MyPatt_int.c>
//::CUT::
/
*****************************************************************************
*
   ▪ Module:  0x116 (tcfa)
   ▪ tcfa_ext_mypatt
 *
   ▪ Description  :
 *
   ▪ Command data : SFCBuffers-generated for cmds_scanf
   ▪ Pattern: u1u4c16c16u4v2* (MyPatt)
 *
   ▪ Answer: based on AnswPatt
 *
 *****************************************************************************
**/
int tcfa_ext_mypatt
    (T_CMDS_HANDLE *p_hdl, int l_cmd, unsigned char *p_cmd)
{
```

```
  int        err = 0;
  unsigned int    l_answ = 0;
  unsigned char   *p_answ = 0x0;
  struct answpatt *answ = 0x0;

  struct { // MyPatt
    unsigned int  flags;                  // u1
    unsigned int  mech;                   // u4
    unsigned char *keyName;                // c16
    unsigned char *keyGroup;               // c16
    unsigned int  spec;                   // u4
    unsigned int  l_hash;                 // v2
    unsigned char *p_hash;                 //
    unsigned int  l_data;                 // *
    unsigned char *p_data;                 //
  } cmd;
  // parse command data
  if ((err = cmds_scanf(l_cmd, p_cmd, "u1u4c16c16u4v2*",
                                          sizeof(cmd), &cmd)) != 0)
    goto cleanup;
  if ((answ = (struct answpatt *)os_mem_new(sizeof(struct answpatt),
                                          OS_MEM_TYPE_SD)) == 0) {
    err = E_TCFA_MALLOC;
```

```
    goto cleanup;
  }
  os_mem_set(answ, 0x0, sizeof(struct answpatt));

  // ... Do Work
  // ... Populate answ
  l_answ = answpatt_length( answ );
  if((err = cmds_alloc_answ(p_hdl, l_answ, &p_answ)) != 0)
    goto cleanup
  answpatt_serialize( answ , l_answ, p_answ );

cleanup:
  if (answ != 0x0)
    os_mem_del_set( answ, 0x0 );

  return err;
}
//::__CUT__::
#endif
```

```
----
```

```
==== Treating the pattern as an answer pattern
```

This is example output, covering points 3 and 4 above. These are predominantly more useful, when 'MyPatt' is used as an answer struct, rather than when used as the input struct.

```
The \#defines `_TCFA_MYPATT_L_KEYNAME` and `_TCFA_MYPATT_L_KEYGROUP` are found in the MyPatt.h header.

[source,c]

.Autogenerated private methods for Serialization, etc

----

/*

  * 01a autogenerated 2019-02-25 using
  * SFCBuffers.pl version 01d,20190225,rip

 *

  * For MyPatt using u1u4c16c16u4v2*

 */

#ifndef _MyPatt_C_INT_1_0_0_0

#define _MyPatt_C_INT_1_0_0_0

#include <tCFA.h>

#include <MyPatt.h>

#include <cmds.h>

#include <load_store.h>

// u1u4c16c16u4v2*

unsigned int mypatt_length(struct mypatt *s) {

    unsigned int len = 0;

    len += 1;                               // u1 > flags

    len += 4;                               // u4 > mech

    len += _TCFA_MYPATT_L_KEYNAME;          // c16 -> keyName (16)
```

```
    len += _TCFA_MYPATT_L_KEYGROUP;     // c16 -> keyGroup (16)
    len += 4;                           // u4 > spec
    len += 2 + s->l_hash;               // v2 -> hash
    len += s->l_data;                   // * -> data
    return len;
}
// u1u4c16c16u4v2*
unsigned int mypatt_serialize (
  const struct mypatt *answ,
  unsigned int l_answ,
  unsigned char * p_answ
) {
  unsigned int err = 0;
  unsigned char * pp_answ = p_answ;
    // flags int 1
    store_int1(answ->flags, pp_answ);
    pp_answ += 1;
    // mech int 4
    store_int4(answ->mech, pp_answ);
    pp_answ += 4;
    // keyName char 16
    os_mem_cpy(pp_answ, answ->keyName, _TCFA_MYPATT_L_KEYNAME);
    pp_answ += _TCFA_MYPATT_L_KEYNAME;
```

```
    // keyGroup char 16
    os_mem_cpy(pp_answ, answ->keyGroup, _TCFA_MYPATT_L_KEYGROUP);
    pp_answ += _TCFA_MYPATT_L_KEYGROUP;
    // spec int 4
    store_int4(answ->spec, pp_answ);
    pp_answ += 4;
    // hash lvfield 2
    store_int2(answ->l_hash, pp_answ);
    pp_answ += 2;
    os_mem_cpy(pp_answ, answ->p_hash, answ->l_hash);
    pp_answ += answ->l_hash;
    // data bkkt 4
    os_mem_cpy(pp_answ, answ->p_data, answ->l_data);
    pp_answ += answ->l_data;
  if ((pp_answ - p_answ) != l_answ) {
    return E_TCFA_SERIALIZATION;
  }
  return err;
}
/* cmds_scanf of mypatt */
unsigned int mypatt_scanf (
  struct mypatt *p_stub,
  unsigned int l_data,
```

```
  char *p_data
) {
  return cmds_scanf(l_data, p_data, "u1u4c16c16u4v2*", sizeof(struct mypatt), p_stub);
}
#endif
----
```

And the respective header file:

```
[source,console]
.Autogenerated Header file
----
/*
   ▪ 01a autogenerated 2019-02-25 using
   ▪ SFCBuffers.pl version 01d,20190225,rip
 *
   ▪ For MyPatt using u1u4c16c16u4v2*
 */
#ifndef _MyPatt_h_1_0_0_0
#define _MyPatt_h_1_0_0_0
/* Defines ------------------------------------------------- */
#define _TCFA_MYPATT_L_KEYNAME 16   /* c16 > keyName > char */
#define _TCFA_MYPATT_L_KEYGROUP 16  /* c16 > keyGroup > char */
/* Declarations for Pattern u1u4c16c16u4v2* for use when
   ▪ MyPatt is an ANSW */
struct mypatt { // answer using MyPatt
    unsigned int  flags;                  // u1
    unsigned int  mech;                   // u4
    unsigned char *keyName;                // c16
    unsigned char *keyGroup;               // c16
```

```
    unsigned int  spec;                    // u4
    unsigned int  l_hash;                  // v2
    unsigned char *p_hash;                  //
    unsigned int  l_data;                  // *
    unsigned char *p_data;                  //
} mypatt;
/* Len computation for Pattern u1u4c16c16u4v2*,
   ▪ Use to determine length of previously populated answ buffer */

unsigned int mypatt_length ( structmypatt * );
/* serialization of mypatt */
unsigned int mypatt_serialize (
  const struct mypatt *,
  unsigned int,
  unsigned char *
);
/* cmds_scanf of mypatt */
unsigned int mypatt_scanf (
  struct mypatt *,
  unsigned int,
  char *
);
#define mypatt_deserialize mypatt_scanf
#endif
```

8.2 Autogenerated Classes in Java/C++ for the Host

For the host, what is autogenerated is OOP classes and support files.

1. Constructors that take a serialized byte array or field values or nothing,
2. Class members that are used in an OOP manner (via getters/setters, etc),
3. Setters that validate data entry (c16 restricts use to 16 char lengths, v1 var-length buffers that limit themselves to 255 bytes, etc) and throw exceptions when code attempts to overstep these constraints,
4. Support for over-loaded input value types (supply byte array or strings for an underlying variable length byte array field, for example),
5. Serialization/deserialization methods,
6. Serialized size determination (max size check for 256Kb, min size, exact size of current serialized size, etc),
7. An overridden `public String toString();` or `debugPrint()` method that is aware of the internals of the class structure (depending on language),
8. Support for exceptions specific to what the compiler is generating (SDKInterfaceSerializationException, SDKInterfaceMaxSizeException, etc),
9. Support for sub-classing of a common functionality super-class (class MyPatt extends SFCInterface { ... })
10. A Test application (MyPattTest.java)

The package shown below is configurable.

Example Host MyPatt.java

```
package com.utimaco.cs2.mdl.tcfa;

/*
   ▪ 01a autogenerated 2019-02-25 using
   ▪ SFCBuffers.pl version 01d,20190225,rip

 *
   ▪ For MyPatt using u1u4c16c16u4v2*

 */
import java.nio.ByteBuffer;
import com.utimaco.cs2.mdl.SDKInterface;
import com.utimaco.cs2.mdl.DeserializationError;
import com.utimaco.cs2.mdl.SerializationError;
public class MyPatt extendsSFCInterface {
    // Begin cmds_scanf
    protected static finalString pattern = "u1u4c16c16u4v2*";
    static final intl_keyName                = 16;    // c16
    static final intl_keyGroup               = 16;    // c16
    byte  flags = 0x0;                      // u1
    int   mech = 0x0;                       // u4
    byte [] keyName = new  byte[l_keyName];    // c16
    byte [] keyGroup = new  byte[l_keyGroup];  // c16
    int   spec = 0x0;                       // u4
    byte [] hash = null;                      // v2
```

```
    byte [] data = null;                        // *
    // Auto-generated constructor stubs
    public MyPatt () { }
    public MyPatt (byte  [] _in) {
      try { deserialize(_in); }
      catch (DeserializationError e) { e.printStackTrace(); }
    }
    public MyPatt (
        byte _flags, int  _mech, String _keyName, String _keyGroup, int
_spec, String _hash, byte[] _data
    ) {
        flags(_flags);
        mech(_mech);
        keyName(_keyName);
        keyGroup(_keyGroup);
        spec(_spec);
        hash(_hash.getBytes());
        data(_data);
    }
    // *Etters
    public byte flags() { returnflags; }
    public void flags (int_in) { flags = (byte  )(_in&0x000000FF); }
    public int mech() { returnmech; }
```

```
    public void mech (int_in) { mech = _in; }
    public byte [] keyName() { returnkeyName; }
    public void keyName (byte[] _in) {
        keyName = maxLenByteArray(_in, 16);
    }
    public void keyName (String _in) {
        keyName = maxLenByteArray(_in, 16);
    }
    public byte [] keyGroup() { returnkeyGroup; }
    public void keyGroup (byte[] _in) {
        keyGroup = maxLenByteArray(_in, 16);
    }
    public void keyGroup (String _in) {
        keyGroup = maxLenByteArray(_in, 16);
    }
    public int spec() { returnspec; }
    public void spec (int_in) { spec = _in; }
    public byte [] hash() { returnhash; }
    public void hash (byte[] _in) { hash = _in; }
    public byte [] data() { returndata; }
    public void data (byte[] _in) { data = _in; }
    public int length() {
      int len = 0;
```

```
        len += 1;                       // flags -> u1
        len += 4;                       // mech -> u4
        len += l_keyName;               // keyName -> c16
        len += l_keyGroup;              // keyGroup -> c16
        len += 4;                       // spec -> u4
        len += 2 + hash.length;         // v2
        len += data.length;             // *
      return len;
    }
    public byte[] serialize() throwsSerializationError {
        int len = length();
        ByteBuffer bb = ByteBuffer.allocate(len);
        bb.put((byte)(flags&0xFF));
        bb.putInt(mech);
        bb.put(keyName);
        bb.put(keyGroup);
        bb.putInt(spec);
        bb.putShort((short)(hash.length & 0xFFFF));
        bb.put(hash);
        bb.put(data);
        return bb.array();
    }
    public void deserialize (byte[] bfr) throws DeserializationError {
```

```
        ByteBuffer bb = ByteBuffer.wrap(bfr);
              int len = 0x0;
        flags = bb.get();                          // flags -> u1
        mech = bb.getInt();                        // mech -> u4
        keyName = new byte[l_keyName];             // keyName -> c16
        bb.get(keyName, 0, keyName.length);
        keyGroup = new byte[l_keyGroup];           // keyGroup -> c16
        bb.get(keyGroup, 0, keyGroup.length);
        spec = bb.getInt();                        // spec -> u4
        len = bb.getShort();                       // hash -> v2
        hash = new byte[len];                      // hash -> v2
          bb.get(hash, 0, hash.length);
        len = bb.capacity() - bb.position();
          data = new byte[len];                    // data -> *
          bb.get(data, 0, data.length);
    }
    public String toString() {
        // The 8 means only display the first 8 lines of the
        // entirely serialized data
        return toString("AutoGenerated com.utimaco.cs2.mdl.tcfa.MyPatt >",
8);
    }
    public String toString(String note, intmaxInitialLines) {
```

```
        try {
            StringBuilder r = new StringBuilder(xtrace(note,
this.serialize()));
            String [] strings = r.toString().split("\\n");
            if (strings.length > maxInitialLines) {
                r.setLength(0);
                for (int  i = 0; i < maxInitialLines; i++)
{ r.append(strings[i] + "\n"); }
                r.append("\t\t...\n");
            }
            r.append("flags = " + flags + "\n");
            r.append("mech = " + mech + "\n");
            r.append("keyName = \n" + xtrace(keyName, 1) + "\n");
            r.append("keyGroup = \n"  + xtrace(keyGroup, 1) + "\n");
            r.append("spec = " + spec + "\n");
            r.append("hash = " + "VarL (v2) field (length: "+ hash.length
+ "): \n"
                        + xtrace(hash, 4) + "\n");
            r.append("data = " + "VarL ( *) field (length: "+ data.length
+ "): \n"
                        + xtrace(data, 4) + "\n");
            return r.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        return "";
    }
}
```

The test code is lengthy and includes connection to a CryptoServer HSM, logging in, etc. It is suitable both to be called immediately, or it can be edited to provide additional/more correct input, either static or from the command line, etc.

Long term plans are to provide "fuzz" testing in line with the pattern requirements, as well as unit testing.

8.3 Autogenerated Lua (CryptoScript SDK)

Output from compiling the pattern:

1. A complete Lua script that implements the pattern
2. The representative ...
3. Additional artifacts for ...

Autogenerated Lua Sub-Function Code

```
----
----
```