Enterprise PKCS#11 (EP11) Library structure

Visegrády, Tamás
Tamas Visegrady/Zurich/IBM@IBMCH
tvi@zurich.ibm.com

2020.02.11.*

*project tree hash ba1d9ae2, commit at 2020.02.11. 08:00:51, document generated: February 11, 2020

Contents

1 Part 1 - Principles of operation 5

2 Sessions and state 8

2.1 Retained content 9

2.2 Object wrapping 9

2.2.1 Functional objects 9

2.2.2 Object integrity checking 11

2.2.3 Attribute-bound keys 11

2.2.4 Transport objects 11

2.2.5 Attributes 12

2.2.6 Boolean attribute listing 13

2.3 Login sessions 14

2.4 Read-only backend 15

2.5 Backend-internal persistent data 15

2.5.1 Recovery of persistent data 16

2.5.2 Persistent data versioning 18

2.6 Usage restrictions, representation and enforcement 18

3 Non-sensitive external structures 20

3.1 Public keys 20

4 Structured state blobs 21

5 Message encapsulation 22

5.1 Transport message structure 22

6 Administration 23

6.1 Administrator setup 23

6.2 Administrator revocation 24

6.3 Administrator authentication 24

6.4 Administrative commands 24

6.5 Administration lifecycle 25

6.5.1 Administrator signature counts 26

6.5.2 Imprint mode 28

6.6 Control points (CPs) 29

6.7 Operational modes (compliance) 30

6.7.1 Compliance mode inheritance rules 30

6.7.2 Compliance mode modification 31

6.7.3 Irreversible attribute settings 31

7 Administration message formats 33

8 Key cloning 43

8.1 Cloning mechanics 43

8.2 Cloning without secret sharing 44

8.3 Cloning through key parts 45

8.3.1 Cloning directly between domains 47

8.3.2 Data transported during cloning 47

8.3.3 Simultaneous import into multiple domains 48

8.4 Key verification patterns 49

8.5 Cloning fundamental assumptions 49

8.5.1 Restricting cloning 50

9 Security assumptions and rationale 51

9.1 Sensitive state storage 51

9.1.1 Authenticated encryption format 51

9.1.2 Authenticated, non-sensitive state 51

9.2 Security rationale 51

9.2.1 Attack scenarios 51

9.2.2 Attack scenarios ignored 55

9.3 Random-number generation 59

9.3.1 Entropy source 59

9.3.2 Deterministic random-number generation 61

9.4 Auditing 61

9.4.1 Audit record construction 62

9.4.2 Audit record, fixed content 62

9.4.3 Audit record, variable content 62

9.4.4 Audit security 63

9.4.5 Audit history 64

9.5 Miscellaneous security-related notes 64

9.5.1 List of algorithm selftests 64

9.5.2 Prohibiting instance identifier export 65

9.5.3 Use of external seeds 65

9.5.4 Test types 65

10 The EP11 host library 66

10.1 The EP11 system interface 66

10.1.1 Target tokens 66

10.1.2 Routing requests 67

10.1.3 EP11 blob specifications 67

10.1.4 Template parsing 67

10.1.5 Size queries 67

10.1.6 Return values 68

10.2 Beyond standard interfaces of the host library 69
10.2.1 The administrative interface 70
10.2.2 Single functions 70
10.2.3 Module and host queries 70
10.2.4 Module management 70
10.2.5 Target group management 71

Appendix **72**

Mechanisms, overview 73
PKCS#11 wire functions 75
Auxiliary wire functions 76
Function overview 77

Part 2 - The wire format **130**

Other references **235**

References **242**

1 Part 1 - Principles of operation

This document describes the EP11 library, a cryptographic service provider (CSP) backend providing services typical of hardware security module (HSM) firmware, and the interfaces it offers to hosts. In a typical production instance, the backend would be deployed in hardware security modules (HSMs) and the host would dispatch to it through a shallow serialization library.

Disclaimer: The information on the product or specific features is not a commitment, promise, or legal obligation to deliver any material, code or functionality.

The development, release, and timing of any features or functionality described for our products remains at our sole discretion.

The EP11 library[1] provides an interface very similar to the industry-standard PKCS#11 API. While the application-visible interface is essentially interchangeable, PKCS#11 manages sensitive data in a server-centric fashion. Unlike most PKCS#11 implementations, *EP11 stores most secrets outside the crypto provider—secure hardware—in wrapped form,* and backend devices are kept essentially stateless [VDO14, Fig.2]. Wrapped secrets are attached to PKCS#11 calls by a host library. Requests are dispatched as self-contained compounds, ready to be processed by any backend equipped with the necessary wrapping key.

Storing state, and attaching it to requests enables EP11 host libraries to utilize multiple backends, offering unbounded potential throughput. Utilizing multiple EP11 modules, the system may offer an arbitrarily high availability, subject to the aggregate RAS of all backends.

EP11 functionality is accessible beneath a PKCS#11 library, which in turn can be a lightweight implementation. The layer on top of EP11 must only provide a mapping between abstract PKCS#11 entities—such as key handles—and opaque state objects. State includes wrapped keys, session state, or other sensitive "blobs". If the handle-referenced objects are replaced by their corresponding wrapped state, the result is self-contained call and may be dispatched directly to EP11. Apart from certain object or session abstractions—which EP11 does not directly support—all functional PKCS#11 calls are directly implemented by EP11, and the host PKCS#11 library does not need to provide them.

Requests and responses pass through the EP11 in the following steps:

1. An application calls a PKCS#11 function, passed to the EP11-aware host PKCS#11 library.

   At this level, functions generally reference objects indirectly. Keys, for example, are referenced through key handles, addressing state indirectly inside the PKCS#11 library.

2. Host PKCS#11 library replaces object references by actual wrapped objects. All references must be remapped, since backing devices do not maintain any persistent objects, and therefore can not resolve a reference.

   Non-symbolic data, such as user input, is passed through unchanged. Parameters of EP11 calls with types defined by PKCS#11 (i.e., those with `CK_...` types) must be directly passed through the PKCS#11 layer.

3. Request parameters and data are encapsulated to standalone, self-contained, TLV (tag, length, value) encoded requests. They are dispatched to the backend designated by the PKCS#11 library.

   EP11 maintains a list of available backends, but the EP11-using host library is responsible for load balancing and traffic control. Traffic routing and transfers are transparent between EP11 components, and are not by themselves security-relevant. *Malicious hosts are assumed to be able to compromise message transport, including routing, but are unable to alter authenticated objects.*

4. The request, once received by the EP11 backend, is parsed. Blobs are unwrapped at this point. If the request is improperly formatted, or there are other inconsistencies, a transport error is returned without further processing.

   Note that certain sanity checks are performed on the host, but everything is checked again in the HSM backend, since anything from outside the HSM is untrusted.

5. After verifying request integrity, and basic consistency of the attached key (state etc.) material, the request is processed as defined in the PKCS#11 specification [PKC04]) Backend-internal code is cleanly separated into a PKCS#11-specific component containing all API-specific knowledge but lacking cryptographic functionality, and a "generic" crypto provider unaware of PKCS#11.

   Raw cryptographic functionality is provided by the crypto provider and underlying crypto engines. In EP11, the crypto provider is a backend-specific port of Clic, the IBM Research/Zürich-originated crypto library.

6. Sensitive output is wrapped before being returned to the host. As described later (p. 9), objects are encrypted and authenticated, with module-resident wrapping and MAC keys.

[1]"XCP", a historical abbreviation, may be used interchangeably to EP11 in documentation or source code.

Non-sensitive output objects, which must not be changed outside HSMs—such as exported public keys—are protected by an HMAC. The dedicated HMAC key is derived from the transport wrapping key, so HSMs in a group of synchronized keys will accept each others MACs.

7. Both sensitive and regular output are packed to a single TLV structure and returned to the host. Reverse transport is assumed to be transparent between backend and host EP11, and it is not security-relevant (it may not compromise blobs).

8. After submitting the response to the channel, all transient storage used is wiped and released. Apart from cached keys, no request data persists in the backend after calls; any observable state updates are returned as part of the response.

   Calls without observable state updates (such as `...Final` calls) simply discard their state, and only output is returned.

   One exception to data persistence is key caching, where unwrapped versions of blobs may remain in the module. This is discussed later. Key caching is API-transparent, as the corresponding keys still need to be provided with every call.

9. The host library verifies response integrity. It returns a failure to the host if the response is not consistent with the request, or other transport errors are detected.

10. Blobs returned to the host PKCS#11 library are saved on the host (but not returned to the application). `GenerateKey` and similar object creation calls produce new state objects; other operations, such as `...Update` calls, change an existing object.

    The host library must index blobs, so that each session etc. may be associated wit the proper blobs.

11. User-visible results of an operation, if any, are returned to the application.

The host-resident blob format combines attributes, including usage restrictions, and key material in encrypted, integrity-checked tokens. PKCS#11 itself allows key transport to separate attributes and data. While we also support the standard PKCS#11 transport methods, we provide an additional transport mechanism to bind attributes and keys. *Our attributes+key transport format allows us to prevent attribute separation or modification, countering a known PKCS#11 weakness.* When imported, keys that have been so transported are excluded from keytransport interaction PKCS#11-compatible formats, i.e., combined transport and less trustworthy imported PKCS#11 objects are intentionally kept separate. For functional calls, where keys are not transported, both object types are available and are not differentiated.

As shown later, host-resident blobs may be bound to *sessions*, which are somewhat different from PKCS#11 host sessions, but follow a similar logic. Sessions are represented by statistically random identifiers (HMAC output, i.e., PRF output); derived from transport keys and user-accessible information. Hosts may further augment user-provided information by job or process identifiers, outside EP11 visibility, such as when such information is logically part of session identity. The derived session identifier diversifies blob encryption, separating sessions into different "cryptographic domains" (sessions' effective wrapping keys will be different). *When a session logs out from a given backend, all key material bound to that session identifier becomes inaccessible, even when stored on the host.*

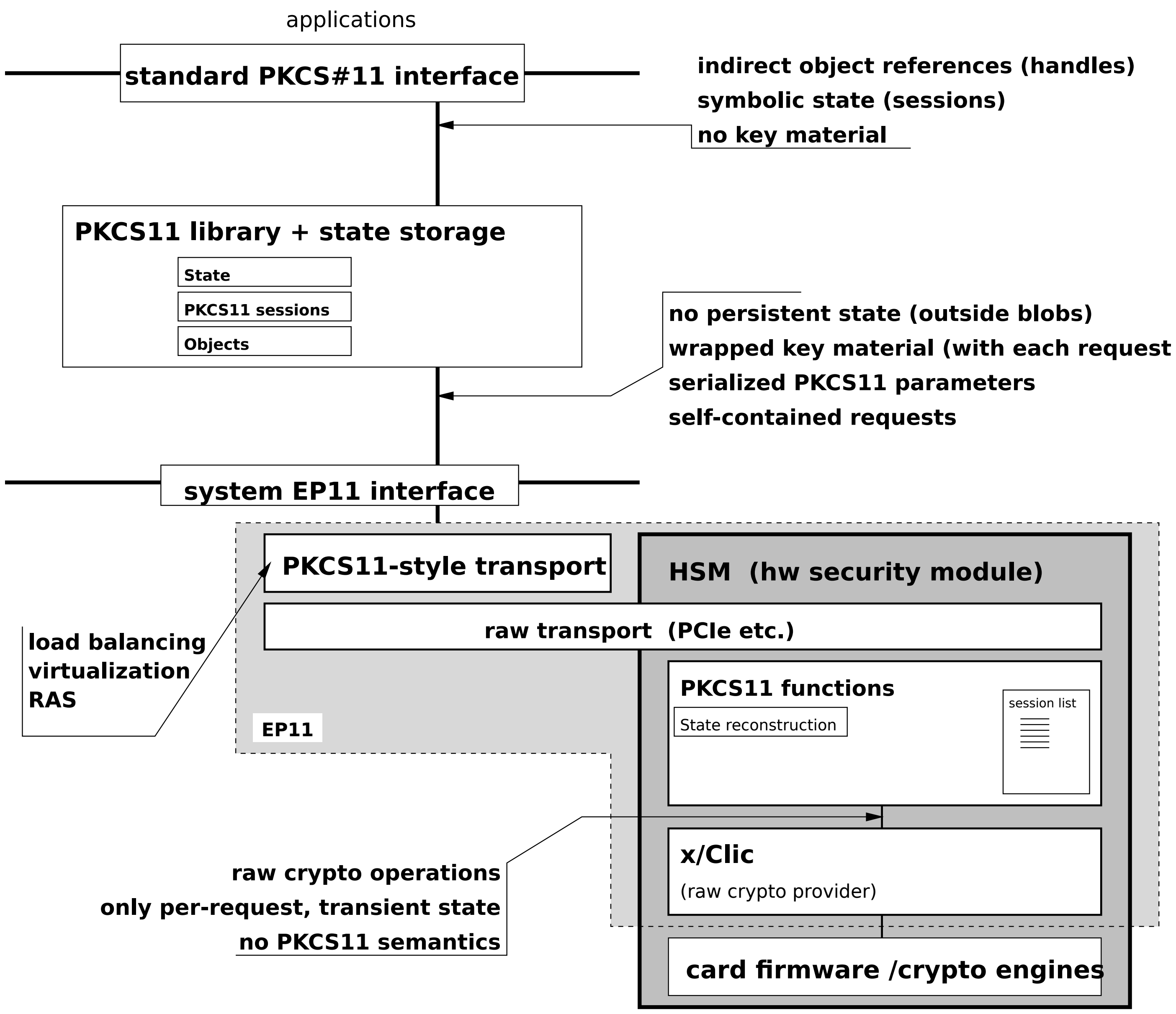
applications
standard PKCS#11 interface
indirect object references (handles)
symbolic state (sessions)
no key material
PKCS11 library + state storage
State
PKCS11 sessions
Objects
no persistent state (outside blobs)
wrapped key material (with each request
serialized PKCS11 parameters
self-contained requests
system EP11 interface
PKCS11-style transport
HSM (hw security module)
load balancing
virtualization
RAS
raw transport (PCIe etc.)
PKCS11 functions
State reconstruction
session list
EP11
raw crypto operations
only per-request, transient state
no PKCS11 semantics
x/Clic
(raw crypto provider)
card firmware /crypto engines

Figure 1: Request transformations

2 Sessions and state

The session model of EP11 differs from PKCS#11 due to the stateless nature of the EP11. PKCS#11 objects are assumed to be unique and have a unique state. When these objects are replaced by selfcontained blobs, the session object is no longer a symbolic reference, rather an instance of one of possible states. Since blobs may be cloned, their multiple instances can appear as multiple sessions. Session destruction also becomes easier, since discarding a blob effectively destroys its state, without needing to notify modules or EP11's host library.

Note that stateless clients have definite security advantages over (more) stateful cryptographic providers: a stateless backend is more resilient against unsafe/incorrect state if that is reachable through the interaction of multiple failures [SCA10, 1.3]. When selfcontained requests are deserialized in their entirety, deserialization and type/content checking may be completely centralized, simplifying analysis of those critical code sections. Similarly, since a stateless backend is in full control of per-request storage, synchronization problems—a non-negligible source of errors [SCA10, 1.3-6]—are simpler to prevent.

An important security advantage of stateless PKCS#11 implementations is that standard API structures may be observed on the wire, but the entire request state may be observed. Serialized objects may be unambiguously tracked, therefore a filter intercepting traffic can identify the specific object/session state even if only observing ciphertext, assuming they have access to past history. *The combination of self-contained requests and standard PKCS#11 functionality allows our backends to be extended with PKCS#11-aware filters if additional access control is expected around the PKCS#11 library* [BCDS15, 3.1].

Since each call uses the object state of a particular instance, replicated blobs behave like individual sessions, starting in an identical state. Obviously, if they are used in different incremental operations, their states will diverge. Unlike PKCS#11, for example, EP11 can perform an `Init` call once, and perform different incremental calls on copies of the returned blob, saving the cost of multiple initializations. This would not be possible in PKCS#11, where each session would need to be initialized after terminating an operation sequence. *Copies* of the same initialized session are logically independent sessions in the EP11 model, diverging only when different data is passed to them subsequently.

Since EP11 objects do not permanently reside in modules, the number of simultaneously active instances is essentially unbounded, limited only by host memory. As a practical consequence, one would not expect EP11 to ever return `CKR_DEVICE_MEMORY`, since module storage would be used only temporarily, released after each request. Similarly, several PKCS#11 specific limitations are not meaningful in EP11 (such as number of parallel sessions or available free memory).

While the state model is different from pure PKCS#11, it can transparently accommodate PKCS#11 for applications that are not aware of EP11's internals. During incremental calls, both PKCS#11 and EP11 update "sessions", either as a true module-resident session, or an external state blob. Unlike PKCS#11 sessions, however, EP11 does not update or finalize state blobs when not necessary. Calls where the useful output is not a blob, such as `Final` or one-pass calls (i.e., `Encrypt`), only output is provided, and the input blob does not get updated to reflect a "finalized" state.

Failing to update host-resident state where not necessary differs from strict PKCS#11 rules, but this difference won't be obvious to a conforming PKCS#11 application. If the application issues traditional PKCS#11 calls, it will freshly `Init` sessions (state) after completing an operation. Since this destroys the previous blob, it will transparently keep working for the PKCS#11 application, even if it may have simply discarded the previous state buffer.

Even if not strictly necessary, calling `Init` for identically initialized sessions is unique where session instances' confidentiality is required. Due to inherent randomness of blob generation, one would end up state objects of identical size, but different actual bits, if the same initialization is performed multiple times. (The single exception to this, clearkey digest states, are not interesting since they do not address confidentiality. When clearkey digest states are wrapped to blob form, they also end up in non-deterministic form.)

Using cloned objects in different sessions does not compromise confidentiality after the first update, since updates change the objects in different ways. As each blob contains a random IV when returned, blob clones updated with identical data streams will diverge at the first update. Depending on host requirement and resource constraints, one may cache initialized, blank objects for frequently used keys, and save the first `Init` calls, if the identity of blank sessions can not compromise security. (An alternative solution, not involving temporary state, is the EP11-specific set of one-pass `...Single` calls.)

One-pass ("`nnnSingle`") calls are unique to EP11; they perform the frequently occurring `nnnInit`/`nnn` sequence for functional calls (`Encrypt`, `Decrypt`, `Sign`, `Verify`, and `Digest`). The purpose of these calls is to skip unnecessary state construction and updates if the operation may be completed in one step. Non-digest calls of this type need an initialized key object and raw data, and return operation output directly, without manipulating state otherwise. Similar to "finalize" style calls, key blobs used during the call are also unchanged, and do not need further administration. (PKCS#11 does not consider the cost of additional transfers, but EP11's state management overhead could dominate shorter operations.)

Since a host library could always restore the blob from its state before the call, destroying an object need not be performed as an EP11 operation. Instead of explicit "dispose" calls, the host would simply need to release the buffer, or to reinitialize it. Both of these is possible without calling EP11, or could be implicitly performed if a new `Init` is issued, overwriting previous state.

Note that most PKCS#11 state management calls are not useful in EP11, where state may be replicated or destroyed simply by manipulating state blobs. Specifically, `C_CopyObject` needs to be replaced by other calls, unless a verbatim copy is needed. Object destruction is unnecessary, since it may be performed by discarding the state blob, and does not need further EP11 involvement. `C_DestroyObject` therefore need not be implemented by EP11 itself, as it gets replaced by a blob manipulation step in the EP11-aware library instead.

2.1 Retained content

For applications where RAS and throughput limitations are not relevant, the API may transparently accommodate backend-resident blobs (*"retained keys"*). In such a setting, calls containing blobs may be replaced with indirect references to HSM-resident persistent state (objects), which does not leave the backend. These keys are useful for high-assurance operations where the loss of any single key may be mitigated against, but key material is of such value, that it must reside within secure hardware for its operational life.

We provide a single mechanism for creating backend-resident objects, which we label *"semi-retained keys"* (since they were host-visible, in encrypted form, at some point during their lifetime). An optional, custom variant of the `Wrap` call transforms a valid token into a backend-internal key object, and returns a handle which may reference it. *We retain all blob metadata, specifically all session information, with the internal object.* Authorization to use the retained object combines knowledge of the handle, and controlling the session which the retained object is bound to. *Retained objects are inaccessible, even with their handle, when the controlling session has logged out of the backend.*

A retained-content provides a service to enumerate its resident objects (in truncated form which does not reveal their handles), and allows one to destroy retained objects. Removal of an object requires cooperation of the object owner, since it requires both the object handle, and the controlling session. *Note that we do not provide a service, even for module administrators, to actually extract a retained object.*

For single-pass services, use of retained keys is transparent. In cases where this is possible, a retained-key handle may double as a "key blob", and gets transparently routed to the module-resident object instead. Retained-object references may be unambiguously differentiated from regular blobs, based on object size and type.

Since retained keys share finite-capacity backends, the number of active retained keys is limited by backend resources. We also set a fixed limit on the maximum number of retained keys for efficiency reasons (we index them internally through a hash structure which is optimized for a build-time constant upper bound). We return the object-count limit as part of the mechanism information corresponding to the "mechanism" corresponding to creation of retained-key objects.

While infrastructure administrator services provide ways of replicating wrapping keys between multiple backends, retained-key objects are intentionally excluded from such replication. This restriction allows our cryptographic users to keep their sensitive retained-key material safe, even if their system is operated by untrusted infrastructure administrators.

Note that enterprise platforms discourage retained objects, and we do not consider retained-key operations necessary. Therefore retained objects are an optional extension, configured as a compile-time option.

2.2 Object wrapping

When storing objects on the host, we use different formats for functional use and interoperability ("transport form"). We mandate users to use transport-form objects when interfacing, with other providers, or when constructing their key material from outside EP11.

Both functional and transport-mode objects are symmetric-encrypted, authenticated blobs, opaque to those without access to their encryption key.

In our descriptions, we do not specify standard PKCS#11 formats, which are also available. Since standard PKCS#11 wrapped keys do not include their attributes, they are kept separate from *"attribute-bound" (AB) blobs*, and may not be managed by attribute-bound keys (2.2.3). By separating PKCS#11 objects from AB ones, we isolate high-assurance key material from the known deficiencies of standard PKCS#11 key management. As these differences are within opaque wrapped objects, AB and PKCS#11 keys may safely coexist within the same provider, and use the same functions, with differences restricted to key un/wrapping. *Since AB and PKCS#11 keys may not un/wrap each other, both types of key hierarchies are restricted to their own types, and therefore PKCS#11 un/wrapping may not compromise the higher assurance of AB keys.*

2.2.1 Functional objects

Sensitive data exported from the module is retained in wrapped form. There is a single, symmetric key for each active domain (outside mainframes, typically a single key); we treat domains as disjoint and refer to "the" key (meaning the domain-specific one). The key may be generated within the module, or migrated from outside (from another module or key backup). It encrypt exported objects directly, or provides the global key which is combined with a per-object virtualization key. Object integrity is verified through an HMAC calculated over object plaintext. The blob HMAC uses a dedicated MAC key, derived from the wrapping key. We use the Encrypt-then-MAC construct to couple integrity and confidentiality [Kra01] [Sma13, 4.3.1].

Exported objects are used in an end-to-end secure channel between backends. Anything between synchronized backends is considered untrusted. Object integrities are verified before use, and outside entities may not modify contents without detection. *This assumption is necessary to be able to use an external, encrypted keystore without trusting anything over the EP11 API.*

Wrapped objects are typed. Each module function verifies unwrapped objects' types, and reacts accordingly. Internally, wrapped objects contain Clic objects, and their type is derived from the Clic type system.

Wrapping steps are performed in the following order:

1. Serialize Clic object to memory, including Clic type information and object markers.

   Object serialization may involve one indirect step, such as when serializing object state and a key—for example, for an incremental signing operation. As with other sensitive state, such combined objects are stored on the host—internally, they are just a different, compound object type.

2. Add/update attributes table, or reuse original (if rewrapping an object)

3. Pad to entire blocks of block cipher

4. Add object-specific virtualization key, if object is virtualized.

   Plain objects start with an all-zero virtualization key, and use the global key directly. They correspond to PKCS#11 "token objects", global entities available to all authorized to use the PKCS#11 crypto service provider (CSP).

5. Add object-specific, random IV

   IVs contain two fixed bytes, which are used to recognize wrapped objects.

6. Encrypt object, skipping initial identifiers (session, TWK) and the IV.

   Use global or virtualized, object-specific wrapping key (i.e., XOR of wrapping key and object-specific mask) if the object is bound to a session.

7. Calculate blob MAC, including leading identifiers (clear) and token ciphertext. Use the TWK-derived MAC key.

Unwrapping reverses the procedure:

1. Verify that wrapped size is suitable for a wrapped object, and that it is indeed a wrapped object (has fixed IV bytes in known location).

2. Construct object-specific wrapping key, if object is virtualized.

   Virtualization key is in the clear. All-zero Virtualized key is used for plain objects.

3. Verify blob MAC.

4. Decrypt encrypted portions of object.

5. Verify that padding, if required, has the proper format

6. Verify that recovered Clic object encapsulation is consistent. This step is redundant (we verified the object MAC first), but we further verify data recovered from such an integrity-checked object.

Type of unwrapped object must be verified by the caller. `unwrap_blob` returns a Clic pointer to the unwrapped payload, since encapsulated content must have Clic object markers, and it will be used as a Clic object. (This is verified during unwrapping.) *The EP11 code itself is not aware of the internal structure of Clic objects.*

Certain functions are polymorphic, accepting both clear and wrapped objects. One example is the sequence of `_DigestNNN` calls, which may digest non-sensitive data using clearkey objects, or sensitive data with wrapped objects.

Wrapped objects are versioned, which permit migration between incompatible backends.

In addition to versioning, the object also stores restrictions within internal attributes. We only store restrictions enforced at a blob level, including non-extractability, wrappability, and similar low-level limitations. Section 2.2.5 describes the actual attribute encoding.

2.2.2 Object integrity checking

In addition to verifying blob MACs before decrypting their payload, deserialized objects within blob plaintext are subject to type-specific checks, during regular read procedures of the underlying CSP. These replacements within plaintext are considered infeasible under regular operations, where host entities generally lack access to WKs, and by implication, derived MAC keys. However, redundant checking is performed to ensure proper CSP operation, even against host-resident attackers with such infeasibly potent error-injection capabilities. *The redundant checks we perform on deserialized objects enable us to prevent abnormal CSP termination* even if host-based attackers attempt to inject them.

Integrity checks for keys of structured types, where bignumber components interact, include sanity-checking parameters for DSA [KR02, 7.2] and RSA [KR02, 7.3]. EC objects are only supported based on specific curves (prime field, NIST [Nat13, D.1.2] or Brainpool [LM10, 3]), therefore curve parameters are fixed, and may not be replaced by host-based attackers.

Note that *the capability of injecting arbitrary plaintext also prevents us from detecting all blob-plaintext modifications,* since an attacker capable of signing arbitrary blob plaintext could also sign any proper signatures.

2.2.3 Attribute-bound keys

Recognizing the security weaknesses of standard PKCS#11 key transport, especially the capability to separate keys and their attributes [Clu03, 2.3], EP11 adds capabilities to manage attribute-bound (AB) objects. The AB property is a proprietary extension, a vendor Boolean attribute which prevents separation of keys and attributes. While the attribute is ignored by other services, keys marked as AB may only be transported in an authenticated-encrypted, proprietary format. Note that none of the standardized formats offer authenticated transport of keys AND attributes, so more secure replacements—those including authentication—are all proprietary [BFSW13, 1] [Ste14, 2] [BCDS15, Appendix B]. The standard PKCS#11 formats, lacking authentication, are flagged as insecure by several vendors [DKS10, 6.1, 6.2].

AB key transport is always authenticated, and all participating keys—key, key-encrypting key, and authentication key—must be attribute-bound. Essentially, for key-transport purposes, AB keys form their own type hierarchies, which are incompatible with non-AB keys. Since the AB property is just an EP11-specific usage restriction, for matters unrelated to key transport, AB objects may be used with functional services using regular PKCS#11 requests. *The attribute-bound property is read-only, may not be changed after key generation.*

Since AB transport is always authenticated and encrypted in one call, it also provides integrity for transported keys, preventing key substitution [Clu03, 2.2]. Obviously, since the signing/MAC keys authenticating an AB-wrapped key may also sign fake AB enclosures, AB transport is not immune to malicious owners of signing/MAC keys, but it is considered immune against other attacks, including corruption. This consideration is documented under our security rationale (9.2.2, page 9.2.2).

While un/wrapping functionality of AB keys is incompatible with standard PKCS#11 services, they are otherwise used identically with functional calls. Therefore, *a typical application not managing its own keys may be easily updated to work with AB keys, and benefit from the higher assurance provided by attribute binding.*

2.2.4 Transport objects

Transport objects are generated for interoperability, and they may not be used for functional calls. We define an attribute-bound transport form which preserves object attributes, therefore incompatible from standard PKCS#11 Un/Wrap calls. Our transport form is an authenticated, encrypted representation of a key, including its access restrictions, providing assurance lacking from standard PKCS#11 wrapped-key representations. Since pure-PKCS#11 key transport does not allow simultaneous attribute transport, AB-keys are by construction incompatible with PKCS#11 ones, while they use the same `Un/Wrap()` functions.

We define transport forms in an implementation-independent way, allowing other implementations to generate or import EP11 key material. *We strongly discourage constructing or parsing functional tokens, even if the controlling TWKs or KEKs may be available, since we may extend or change internal representations.*

Note that regular PKCS#11 objects support the usual PKCS#11 un/wrapping mechanisms. AB objects must be generated by AB-keys, separating PKCS#11 blobs from AB ones: only AB KEKs are allowed to encrypt AB keys, and AB signing keys must sign them. Complete separation of AB keys from regular PKCS#11 ones allows us to maintain both AB and PKCS#11 keys within the provider, while maintaining them in different trust domains.

Transport forms intentionally restrict the choice of algorithms, and only support fixed combinations:

1. Symmetric encryption must be TDES or AES; other symmetric KEK types are rejected.
2. If an asymmetric KEK is used, the symmetric encryption is always AES/256, with the key being transported by an algorithm corresponding to the KEK:

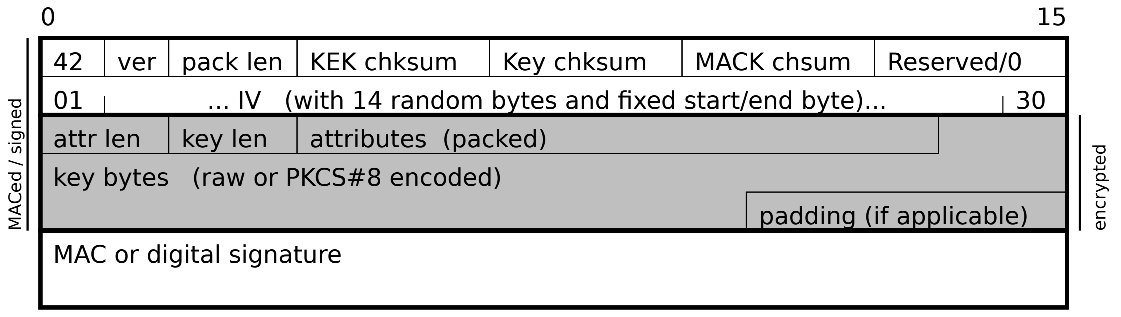

Figure 2: Structure of transport-encoded keys

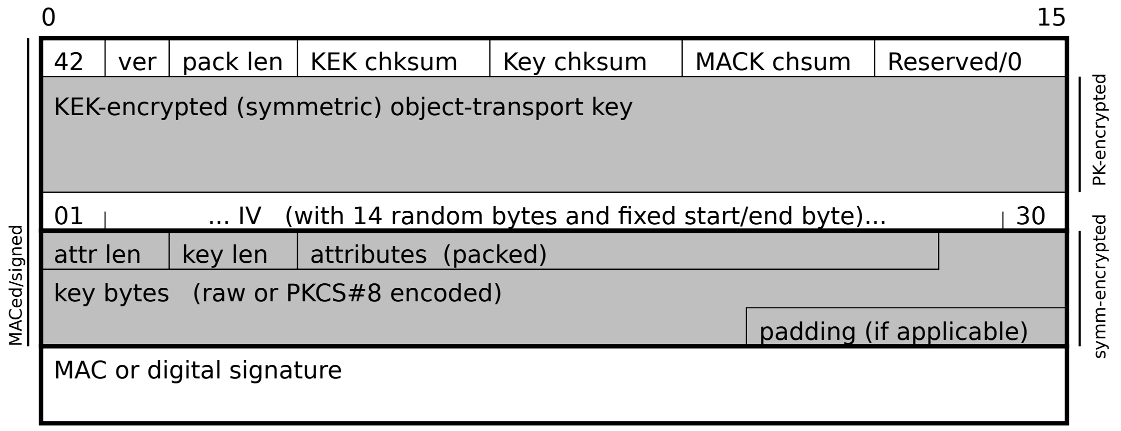

Figure 3: Structure of transport-encoded keys with asymmetric KEK

- RSA keys encrypt with OAEP
- EC keys move the symmetric key through ECAES [SEC09, 5.1], in a static-ephemeral scheme similar to RSA [BJS07, 6.2.2.1]

3. Symmetric MAC keys generate/verify HMAC/SHA-256 signatures
4. Asymmetric sign/verify keys use
   - RSA keys use PSS with SHA-256
   - EC keys use ECDSA with SHA-256 hashes

Please see the wire specification for the layout of AB-wrapped objects.

Transport form is only defined for key objects; session objects such as incremental encrypting streams are never transformed to or from AB form.

2.2.5 Attributes

Host-resident objects, both in functional and transport form, feature three attribute categories, in order of decreasing frequency:

1. Boolean attributes, encoding binary settings, the vast majority of attributes we use.
2. Numeric attributes, stored as fixed-size integer *types* and *values*. Note that fixed size encodings are possible for most relevant PKCS#11 attributes.
3. Variable-length attributes, either variable-length scalar values or recursively defined attributes.

We store the three attribute types in a variable-length, fully specified field. The packed format is used in both functional and transport form, with two additional attributes in transport form (which are implicit in functional blobs, and therefore not stored within the attribute area). We use an encoding which allows single-directional parsing (one pass, from lower to higher offsets). In both functional and transport forms, a separate—redundant—length field prefixes the attribute field, allowing reading to skip the attribute field completely. See the wire description for encoding details.

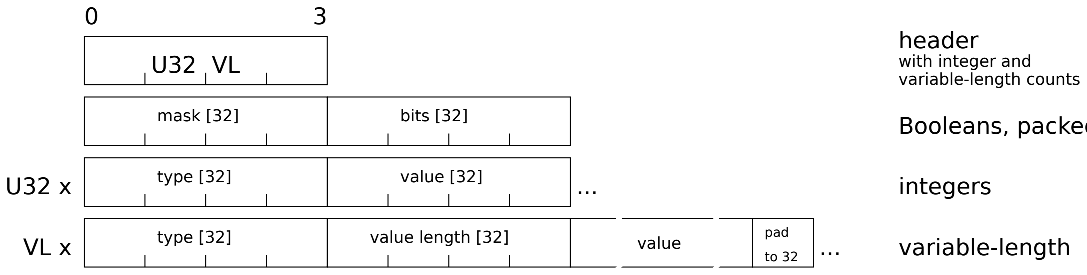

Figure 4: Attribute encoding in functional and transport encapsulation

All attributes may be queried; see `GetAttributeValue()`, our adaptation of the PKCS#11 original.

Attributes may be changed through `SetAttributeValue()`. Currently, only Boolean attributes may be modified, and modification requests containing numeric or variable-length attributes are rejected.

2.2.6 Boolean attribute listing

The following attributes match their PKCS#11 equivalents, set if the object has the equivalent `CKA_...` bit. The PKCS#11 attributes not on this list are ignored by the backend, and have no compressed—packed Boolean—representation.

1. `EXTRACTABLE`, may not be set once removed
2. `NEVER_EXTRACTABLE`, read-only
3. `MODIFIABLE`, preventing modification once removed
4. `SIGN`
5. `SIGN_RECOVER`
6. `DECRYPT`
7. `ENCRYPT`
8. `DERIVE`
9. `UNWRAP`
10. `WRAP`
11. `VERIFY`
12. `VERIFY_RECOVER`
13. `LOCAL`, set for objects which were generated internally, not imported. Read-only, not influenced by host.
14. `WRAP_W_TRUSTED`, set for objects which are only allowed to be wrapped by keys with the `TRUSTED` attribute.
15. `TRUSTED`, set for KEKs which are allowed to wrap objects flagged `WRAP_W_TRUSTED`.

    Note that setting this attribute is not access-controlled, and depends on host cooperation, if administrative controls allow its unauthenticated state.

Note that `VERIFY` and `ENCRYPT` are implicitly added to public-key objects which support this operations, because public-key operations are not restricted. Since public-key objects are stored in authenticated cleartext form and operate on host-visible plaintext, the host may perform those operations on its own. Therefore, verify or encrypt capabilities on host-visible data are effectively always available, which we recognize with these defaults. The related `WRAP` functionality of public keys, since it operates on sensitive—and therefore not host-visible—data, is not implicitly provided.

In addition to standard Booleans, the following extended ones are supported:

1. `NEVER_MODIFIABLE`, read-only, set for objects if there were created or imported to be non-`MODIFIABLE`.

   We define `CKA_IBM_NEVER_MODIFIABLE` as the corresponding attribute.

2. `RESTRICTABLE`, a weaker form of non-modifiability. Objects with this attribute set are still modifiable, but they may not have attributes added, only removed.

   The primary use of restrictability is to create multiple versions of restricted-use objects, such as sign-verify pairs, which must not encrypt/decrypt. Creating objects with the superset of all final ones and `RESTRICTABLE` allows one to gradually build the related versions of the object—removing attributes in multiple steps—without the capability of creating with less restrictions. In such a setup, one would probably set the final versions to non-`MODIFIABLE`.

   The corresponding vendor attribute is `CKA_IBM_RESTRICTABLE`.

3. `ATTRBOUND`, set for objects which may only be transported during attribute-bound (AB) key transport, in non-standard-PKCS#11 form, which combines attributes and raw keybytes. A read-only attribute. See also attribute-bound objects elsewhere (section 2.2.3).

   AB keys must be wrapped and authenticated by other AB keys: the KEK, the MAC/signature key, and the transported key must all be attribute-bound. For functional calls not related to un/wrapping, AB and non-AB keys may be freely mixed. Note that AB-wrapping mechanisms do not need to provide attributes during key transport, since all attributes are included in the encrypted and authenticated package.

   The corresponding vendor attribute is `CKA_IBM_ATTRBOUND`.

4. `USE_AS_DATA`, set for keys where raw key bytes may be used as "data" of some cryptographic operation, such as hashing (`DigestKey()`) or key derivation (`DeriveKey()`). This restriction further controls key-based operations which do not involve key migration, therefore, are not controlled by `EXTRACTABLE` or transport-related control points.

   The corresponding vendor attribute is `CKA_IBM_USE_AS_DATA`.

2.3 Login sessions

As a special case of session management, backends may store a list of *logged-in sessions* to map (groups of) host entities to module-visible identifiers. These "login sessions" are an exception to backend statelessness: they have been added intentionally, to allow host-originated revocation or activation of groups of keys, potentially entire keystores. Functionally, login sessions resemble PKCS#11 PINs, mapping host-visible entities to EP11-visible user identities. Since virtualized hosts may have different understanding of what constitutes a single identity—such as: do applications in separate partitions belong to different entities—host assistance may be necessary when translating between application and EP11-visible PINs, as described below.

Login sessions are tracked through fixed-size identifiers, each derived from a user-provided PIN, and a corresponding, optional nonce. *The session-ID derivation process calculates a MAC—HMAC/SHA-256—over this input with a fixed key, effectively calculating a hash of PIN and nonce.* (Note: since this functionality is embedded within functional calls, no stronger authentication is achievable beyond proof-of-possession. See also the security rationale.) The MAC on the returned session identifier—i.e., "PIN blob"—may be verified by any other module.

The two session-management commands, `Login()` and `Logout()`, must be called on a per-module basis, since they manage per-backend state. Hosts supporting single sign-on or comparable techniques may automatically log in a set of sessions, if a new module appears in the system. Once the same session has been established within a newly added module, it may access any session-bound object even if created within other modules.

Host entities sharing the same PIN and nonce are, for practical purposes, the same identity even if multiply instantiated—they will derive the same, identical session identifier. Especially in virtualized environments, host libraries are expected to augment PINs and nonces, such as appending "application/partition identifiers" or other differentiating information to the parameter, if multiple instances represent different identities. *Since backends lack host context, such as job, process, or application identity, all relevant diversification must be embedded to PIN/nonce before calling session-management services.*

Session identifiers need to be retained by the originating user, as they are identified through proof-of-possession. Therefore, while a session identifier is insufficient to reconstruct the original PIN—and optionally, nonce—it still needs to be controlled by the owner, as it is used to demonstrate ownership. Note that session identifiers are effectively hashed versions of the originating input, containing neither PIN nor nonce directly, therefore they are assumed to be non-reversibly mapped, and do not endanger the originating PIN/nonce when "exposed" [Mur08, 3]. This protection resembles the traditional security assumptions of device-bound credentials, allowing "loss of device" to prevent disclosure of the originating user information [Goo08, Auto-login].

Session identifiers may be supplied to commands generating or importing new keys, and binds the created objects to both the controlling WK and the session. *Blobs contain a part of the controlling session in their clear header, which by itself is insufficient to create new objects bound to the same session.*

Since EP11 login sessions do not completely map to PKCS#11 token/slot-level sessions, application-level management functions are not implemented. Host libraries are expected to map host-visible applications and entities to sessions, without passing `CloseAllSessions` or similar PKCS#11 calls to backends.

Login sessions are orthogonal to WKs, and are unaffected by WK removal or rollover. Obviously, session-bound blobs expire if the underlying WK changes, but the controlling session may then be used to create keys for the updated WK.

Session checking is centralized: blobs—including session-bound public keys—are rejected if the controlling session is not present in the targeted module. Individual PKCS#11-visible services therefore need not interact directly with session checking, although they may return an inherited error, indicating that the attempted blob is controlled by a missing session.

Note that `Logout` requires the entire session identifier (but not the originating PIN or nonce). Therefore, observing a session-bound blob—containing only a part of the identifier—is insufficient to log out the corresponding session. Similarly, intercepting `Logout` is insufficient to recreate the same session by unauthorized users. If the session identifier is lost, and may not be reconstructed, the session is effectively locked within the affected module. For such emergencies, or forced removal of sessions, an administrative command is available to trim the session list. Note that while they can force removal of sessions, *administrators are assumed to be incapable of establishing the same session* without access to the originating `Login` credentials. (Obviously, administrators able to access user memory can impersonate users at will, therefore they are able to re-establish their sessions.)

As with other functionality, all administrators are more privileged than any PKCS#11 entity, therefore allowing forced removal of a lower-privileged entity—i.e., a PKCS#11-level login session—is consistent with our security assumptions. Note that by lacking the ability of logging in a specific session identifier, *even administrators may not impersonate a user* unless they can intercept the transmitted PIN/nonce. Since any host entity capable of such interception may impersonate the originator, EP11 administrators have no special advantages to launch such attacks. Obviously, against host administrators capable of observing an entire user session—such as full-machine administrators capable of accessing memory of the user partition—sessions offer no security.

The number of available sessions is finite, with a count limit reported to the host. As with other, similar limits, we expect to increase the limit in future releases, but do not publish the limit as an official constant, relying on automatic discovery instead.

Note that sessions form a flat system currently, and therefore grouping etc. of sessions is not supported. However, session-identifiers have been defined with some expansion-capability—see the wire section for details—and therefore hierarchical grouping, or other session-management extensions may be added in the future. *We currently foresee hierarchical sessions, such as groups of sessions, as a feasible extension.*

2.4 Read-only backend

When the backend is in read-only mode, persistent backend state may no longer be updated. Objects belonging to host-resident databases are still updated, so operations .

The following operations are prohibited by a read-only backend:

1. Retained keys may no longer be loaded. Existing keys remain available.
2. Sessions may not be added to removed by the backend. Existing sessions remain available.
3. State-changing administrative operations are prohibited. This includes all signed commands, and the few unsigned, state-changing ones (such as WK commit).
4. The audit subsystem is stopped, as it may no longer save its state to persistent storage, which is a prerequisite of event-chain consistency. Operations which would generate new audit entries, such as key generation, are rejected.

Note that a PKCS#11 vendor-extension return value indicates rejection by a read-only backend (`CKR_IBM_READONLY`).

Read-only mode may be imposed on our backend code by the underlying infrastructure—such as an ongoing *concurrent firmware update* which prevents filesystem access. In addition to such environmental conditions, test additions allow us to set read-only mode with diagnostics builds (to allow testing checks related to enforcement of read-only mode).

2.5 Backend-internal persistent data

Persistent data stored by the backend is not directly visible over the wire interface; internally, they are files in BBRAM or flash-based filesystems. Since our target environment in HSMs may be powered down at arbitrary times, backends must be prepared for atomic file updates, in addition to any file-integrity checking. Since the backend stores multiple, partially coordinated files, there is an additional requirement for consistency of multi-file updates.

Incremental writes for persistent files are not supported: each file-write replaces previous contents by replacing the entire file in a two-stage commit. Updates to files are individually atomic by design. This “whole-file write and replace” update process, without updating files in-place, substantially resembles the update process of state-of-the-art logging filesystems [RBM13, 3].

File integrity is provided by including a cryptographic hash, added and removed transparently by persistent-storage functions. Since partial file access is not supported, each file `write()` and `read()` may transparently add or verify the file-hash, without exposing it to the calling function. If the calculated hash of non-hash file bytes does not match the hash section of the file, contents are reported as corrupted, leaving the caller to respond in a context-specific manner. If the system is restarted with file contents reflecting partially committed writes [PCA+14, 2.1], we assume to detect corruption due to lacking a proper hash field—therefore, by providing our own file-hashes, we expect to run safely even on filesystems with no atomicity guarantees. *Our only atomicity expectation is the POSIX-mandated atomic file-replacement afforded by calling `fsync(2)` on the file, then on its directory* [Man14] [Gro97]. This double-synchronized replacement sequence is de facto atomic on relevant filesystems [PCA+14, 4.4.3] even if not all implementations fully comply with POSIX-mandated atomicity rules.

We generally do not distinguish between transient and persistent failures: *we attempt to "repair" module state* by ignoring any invalid data, and recreating the affected files in a known-good—generally, factory—state. While guessing or recovering partial data is used by certain filesystems when encountering corruption [PBA+05, 3.3], *we categorically reject any data found in corrupted files*. Note that since HSMs are expected to be zeroized at any time, HSM-aware host code is already expected to gracefully handle new HSMs showing up in "factory state".

File corruption is reported differently from a missing file, so an initial module setup—upon first boot—may be differentiated from finding a corrupted file. The first successful startup in a factory state—i.e., no previously existing files—creates all persistent files, even if in an initially-empty state; subsequent module startup detects and log the lack of expected files.

Certain persistent structures storing mixed types of data may be further split into multiple files. As an example, administrator certificates require separate de/serialization, and are expected to change infrequently. At the same administrator level, transaction counters reside in fixed-length structures, are trivial to de/serialize, and will be updated during each state-changing command. These structures are then stored in two files, and their cross-consistency is separately checked, see "Recovery of persistent data" below.

Note that designated transient files, such as test procedures' intermediate state, are excluded from hashes and transparent integrity checks. These files are either absent from production, or they are discarded if the module is ever restarted—and may feature their own integrity checks. For backend purposes, they effectively replace transient memory structures, not persistent storage, even if they are stored in a filesystem. Production code only stores intermediate files during state import and export, which are covered by digital signatures in their entirety, so the lack of integrity checking of partial files is irrelevant.

2.5.1 Recovery of persistent data

While files are individually integrity-checked in an all-or-nothing fashion, logical dependencies between multiple files are resolved when all or some of them are read back and verified. Interconnections are generally caused by the backend separating variable-sized and infrequently updated structures from inflexible, fixed-size, more frequently updated ones:

Administrative certificates and the corresponding transaction counters etc. form a single logical unit, but are separate files. Since the fixed-format structure also includes attributes relevant to certificates—such as: administrative thresholds controlling their use—these two files must remain consistent.

Administrative data is split both at the module and domain levels. Domain data is serialized in a single structure, so there is a single domain structure for both certificates and other settings.

(Semi-)retained keys and their attributes, including usage counters, form a single logical unit, but are stored in two files. Only attributes are expected to be regularly updated, but the files must not be separated or mismatched.

Note that only certificate-management services need to modify both administrative certificates and other attributes within a single administrator command. During these updates, the only attribute changing within attributes is a transaction counter: no command allows simultaneous modification, as an example, of signature thresholds and certificates by a single command. *If we allow the transaction counter to be updated before the certificate list, there is no need to add proper transactions around commands updating both structures.*

We special-case administrator certificate-list updates to allow consistent completion without proper transactions, but still preventing replay of administrator commands. allowing the host to recover from the pathological case of a module restart interrupting this two-file update:

1. When accepting a command changing the certificate list, verify that it advances the transaction counter from $M < N$ to N, while changing the certificate list from $C1$ to $C2$. By implication, this command will not be accepted once the counter reaches N or higher. (Note that the signed command does not include M or $C1$ directly, containing only N and the difference of $C1$ and $C2$. We include them to clarify context.)

   The $C1 \rightarrow C2$ and $M \rightarrow N$ state changes need to be stored in separate files, "simultaneously" during command processing. This is the original restriction which raises the question of transactional updates of two files.

2. Update certificate list and transaction counters in memory. Both structures differ from their saved versions at this point: memory contains $(C2, N)$ with the filesystem still at $(C1, M)$.
3. First, commit the updated attribute structure to file. This "inconsistent" state advances the transaction counter only, but does not yet update the persistent certificate-list copy.
4. Upon successful commit of attributes, persistent structures contain $(C1, N)$. If the same update command is received, it will be rejected, since the transaction counter is no longer below N. At the same time, the file-persistent state still contains $C1$, so the system will be successfully started in the $(C1, N)$ state if interrupted at this point.
5. Commit the certificate list to file from memory. Upon successful completion, filesystem state advances to $(C2, N)$.

   After this commit, the module reached $(C2, N)$ as a final state.

Note that even the intermediate persistent state is consistent, even if it has yet failed to update the administrator certificate list. The only problem with the above process is that a module, when in the intermediate state, may not reapply the originating command due to the transaction counter advancing first. *If the system is interrupted in the above $(C1, N)$ state, administrators need to sign the command again, with an updated transaction counter $X > N$, and re-submit to the same module.* This limitation is acknowledged and documented; we already require administrators to react to the certain recovery steps, and do not resolve those within the module. *Our state model simply special-cases the intermediate state as a possible one, acknowledging that these updates pass through three states atomically, unlike most others which only force one atomic transaction.*

Since persistent structures form multiple, hierarchical groups, recovery from an inconsistent/partial set of files is possible, at the cost of removing some invalid files (Fig. 6):

1. Logged-in sessions (section 2.3) have only functional-level relevance; they are discarded if their backing file is corrupted. This loss has an influence at the PKCS#11 level (i.e., `Login()` state), and must be recovered from the host, but it has no other module impact.
2. (Semi-)retained keys and their usage restrictions form a single, functional-level group. If any of them is corrupted, the other one is discarded, and host code must recover from the loss of files. No other persistent structure is affected.
3. Domain-level structures separate administrator certificates, from all other—frequently updated—administrative data, in a single structures collating information for all domains. If either file is missing or corrupt, the other one is discarded if it is present.

   Domain-level recovery zeroizes all domains, and any functional data which has depended on WKs or other functional-level setting (sessions, retained keys). Module-level structures are unaffected.
4. Module-level structures (administrator certificates, all other administrative data) must be both present. If either one is missing or invalid, the other one is discarded if present, and the module enters administrative imprinting, just as it started in factory state.

   Card-level reinitialization forces removal of domain-level administrative structures, and any additional implied changes. The audit subsystem, since it is not influenced by any administrator, is unaffected. Any transaction-marking file will have been removed by the time of administrator recovery, so transaction files are unaffected.
5. Audit state structures, if ever corrupted, are discarded and recreated in a new, randomly-instantiated audit root (see section 9.4.2). The newly created audit-event root logs the detected file corruption and recovery as its first event, but no other action is taken.

   Since the audit subsystem operates independently of any other module subsystem, its file recovery does not change any other persistent file.
6. Transaction identifiers are only stored during transactional operations, such as when marking an entire filesystem being updated during state import. (The transaction identifier is simply an integer, stored in the filesystem, identifying the target final state.) These files only exist before transaction start and completion, and may be encountered during startup if the module has been restarted before commit.

   Transactions are resolved as the first action when a transaction file is found, so they may affect other files.

   A malformed transaction file, when encountered, is interpreted as a catastrophic failure. All persistent files, except any existing audit-state file, are removed in such cases.

Audit events are generated for all forced file removals. The host may also observe the loss of structures through regular queries, but no other notification is provided when files are removed due to corruption.

2.5.2 Persistent data versioning

While backend-internal file formats are not exposed through the wire interface, and are not documented here, IBM internal documentation exists for them. Since these structures are designed to be migrated to future formats, we require versioning and well-defined structures for all of them.

2.6 Usage restrictions, representation and enforcement

Restrictions on the use of attribute-equipped blobs, which includes all keys and incremental-session state, are enforced in a hierarchical set of checks. Allowing or rejecting use of any object is controlled by the combination of all restriction categories (Fig. 5), allowing only the subset of objects allowed by all:

1. CPs of the responsible domain for all non-administrative requests, and a few domain-specific administrative-related ones

   While CPs restricting keytype/strength/mode form clearly defined groups, most CPs are not so categorized. These "other" CPs impose usage restrictions on very diverse points of backend control flow (therefore the highlight in Fig. 5 with no direct connection: these CPs interact with almost all other shown attributes/restrictions).

   CPs may represent other infrastructure-derived restrictions, such as those imposed on the hosting platform (see FCV). These architecture-imposed restrictions are processed as regular CP-controlled ones are.

2. Administrative attributes derived from per-domain CP setup, or their card-level aggregated equivalents. Fig. 5 shows this relationship, when domain/card compliance attributes are derived from CPs.

   Note that compliance attributes, as an example, are a condensed representation of multiple CPs. They are provided as read-only attributes to show compliance—or lack thereof—with security standards or regulations (see 6.7)

3. Key size, type, initialization state, usage restrictions of the blob-internal key/state, as recovered from the blob.

4. The list of sessions maintained adapted forms of PKCS#11 commands (2.3)

5. Restrictions on key size, type, initialization state, operational mode etc. imposed by the PKCS#11 API. These restrictions are encoded in control flow, are not runtime-controlled, and are inherited from [PKC04], adapted to our backend.

   API-imposed restrictions specify tuples such as: `Encrypt()` requires an encrypt/state object output by `EncryptInit()`, which has not yet used by an incremental `EncryptUpdate()`. The base key initalizing encryption state must have been an encryption-capable keytype, and had its `CKA_ENCRYPT` attribute set. Data passed to `Encrypt()` may have size/format restictions based on algorithm/mode etc.

As shown in Fig. 5, the different types of restrictions force decisions based on different properties of each object:

- the availability of PKCS#11 services may be prohibited by CPs. As an example, use of the `WrapKey()` service is prohibited if CPs of the responsible domain prohibit any kind of key export, even before checking whether the supplied blob is `WRAP`-capable

- services generally accept only a subset of objects, which must be of the given mode; see `Encrypt()` example above.

  Object checking combines base functionality (`Encrypt()` requiring state initialized for encryption), past history (incremental and single-call `Encrypt()` calls may not be mixed), and other similar calls.

- both keytype and cryptographic strength of the object must be permitted by the current CP setup.

  Key type checks implicitly include algorithm-category checking: use of any kind of private key is separately controlled from use of elliptic-curve keys (or even use of specific categories of elliptic curves, see the defined CPs).

- the entire set of object attributes must be consistent with the CP set of the responsible domain. As an example, if the CP setup prohibits use of keys capable of both en/decryption and un/wrapping, en/decryption and un/wrap-related object attributes are cross-checked (such as: reject attacks mixing keys and data through encrypt+unwrap-capable dual-use keys [Clu03, 4]).

  Since these interactions involve entire sets of attributes, they are not separately shown in Fig. 5.

- if an object is session-bound, its controlling session must be active (present) in the targeted backend

  The PKCS#11 notion of a "private object"—those bound to logged-in PKCS#11 sessions—map to EP11 sessions, therefore we show a connection between sessions and PKCS#11-derived restrictions.

- for functional use, the compliance mode of the object must match the then-current administrative compliance attribute. (This, in turn, is a condensed representation of the full set of CPs.)

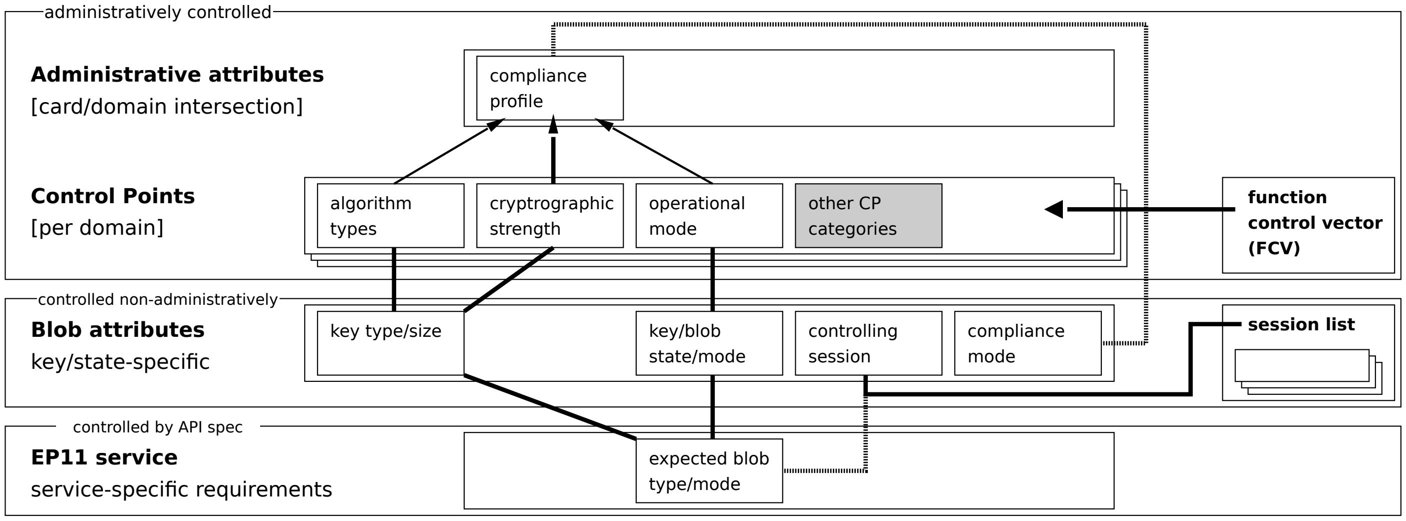

Figure 5: Usage restrictions: hierarchical enforcement

Objects must be allowed by each of these restrictions to become eligible for use. Failures are reported as a combination of standard PKCS#11 errors, such as *CKR_KEY_TYPE_INVALID*, and some IBM-extended ones, mainly for CP-mandated errors (which have no PKCS#11 equivalent).

To somewhat simplify policy/setup-induced error reporting, we distinguish between policy rejections which *may* be disabled by CP changes, and failures where the responding backend can not be configured to accept that object. As an example, a future backend removing support for deprecated algorithms would reject existing blobs of this deprecated types with the latter policy rejection error.

Most of restrictions-enforcement is centralized: the backend uses a single `unwrap_blob()` service, which is responsible for all context-independent checking. Since most restrictions may change at runtime, all checking is applied against the then-current setup. *As with Unix file-permission checking, once an call is approved, it may be allowed to complete, even if the setup changes after the check, during execution* (which, under regular operations, does not generally happen).

The specific order of checking restrictions is not specified. We generally minimize unnecessary computation, therefore restrictions which may be evaluated on plaintext-visible information—such as mechanisms—are applied before those dependent on blob-plaintext (such as blob-internal stream state). *We intentionally do not specify the specific order of evaluation, since that may need to change when restrictions are added in the future.*

We note here that a significant complexity of our regression suite is present only to ensure that specific usage-restriction errors are encountered where expected. While constructing error cases is not very complicated, ensuring that CP and other setup lets those invalid requests through to the targeted check, without triggering other errors, is quite complex. Recognizing the futility of completing this manually, we constructed many of these compound conditions based on conditionals derived by static analysis tools—note that BEAM, the IBM-developed static analyzer constructs compound statements immediately suitable for turning into such erroneous conditions [Bra00, 7] [BBS06, 6].

3 Non-sensitive external structures

In addition to wrapped data, certain operations export state in cleartext objects. These objects contain public keys and related non-secret information, which need not be stored in wrapped objects.

3.1 Public keys

Public-key related data is exported in three related formats, all based on standard `SubjectPublicKeyInfo` *(SPKI) ASN.1/DER structures*[PHB02]. In addition to SPKIs, minimal additional data is added to turn the SPKI into a native EP11 "object":

1. WK identifier, showing the controlling WK
2. Session identifier, allowing the SPKI to be bound to a logged-in session
3. Salt, generated within the module. The salt prevents host entities from generating a MAC of entirely user-specified content.
4. Object attributes (see 2.2.5)

SPKIs encode both key type and public parameters. For RSA keys, modulus and exponent are the only two parameters, which are prefixed by an "RSA" object identifier. Such compounds (ASN.1/DER SEQUENCEs) are widely recognized as standard containers for RSA public keys. EC SPKIs contain more fields, but are similarly standard formats. (Note that the backend only contains a number of fixed EC curves, and currently does not support custom domain parameters, relying on curves specified through object identifiers.)

Plain SPKIs are used where a mechanism is available, and the public key does not need to be imported to an EP11 entity. Digital signature verification, is possible with plain SPKIs, as the mechanism is available as a separate parameter. (Verify calls are actually polymorphic, accepting an SPKI with a MAC, if that is provided.)

Integrity-protected SPKIs with MAC, without an embedded mechanism, are needed where one may not use a public key without trusting it, specifically wrapping other keys. *Key wrapping requires wrap-capable public keys, unless the transported key allows transport with raw SPKIs.* For keys *not* labeled as "wrap with trusted"—those without the `WRAP_WITH_TRUSTED` attribute—`WrapKey` accepts public key KEKs (i.e., SPKIs) that are not MACed. (Setting a key as trusted is a standard PKCS#11 operation. We support it, with CP-based access control, as it requires host cooperation.)

Operations where one needs to bind a mechanism to a public key use a variant of MACed SPKIs. While these structures are primarily useful to retain combined key/mechanism as a single unit (such as between `EncryptInit` and Encrypt calls), they also protect the "session" from modification. Note that unlike symmetric `Encrypt` operations, one does not need to maintain state for RSA Encryption, as supported RSA/EC encrypt mechanisms do not allow incremental operations [PKC04, Table 34] [PKC15a, 2.1].

In combined structures, the mechanism is stored as an additional field within an SPKI compound, storing the mechanism in addition to the SPKI. Backend functions verify that the enclosed mechanism is consistent with the intended operation. RSA encrypt and sign/verify accept the same structure for mechanisms that can both `Encrypt` and `Verify`.

Standard ASN.1/DER parsing can trivially retrieve the length of EP11's embedded SPKI. Due to inherent limitations—limited range of supported public-key sizes—SPKI tag/length fields always fit within the first four bytes of the SPKI, which may be used to quickly verify object length.

Due to the nature of ASN.1/DER parsers, SPKIs with an appended MAC are usually accepted and parsed without additional effort, as the DER header uniquely encodes the length of the SPKI. Structures starting with a mechanism must skip that field to parse the SPKI.

Since authenticated SPKIs include attributes, in addition to keytype/size checks, their attributes are cross-checked as regular blob attributes are (see 2.6).

4 Structured state blobs

Table 1. summarizes interactions of object states and groups of functionality. The following state types are stored on host:

1. *Mechanism and SPKI*, protected by an HMAC. These objects are used for mechanisms where a public key is sufficient (such as digital signature verify).

   These state objects are stored in clear on the host (they do not contain sensitive information, keys, or state).

2. *Digest state (including mechanism) and corresponding SPKI.*

   Digest+SPKI is stored where a public-key operation needs state (i.e., incremental digital signature verify). These objects are encrypted, as the digest state contains plain data bytes.

3. *Mechanism and private RSA key.* Used for one-pass ("stateless") RSA decrypt or sign, where no hashing is involved.

   These operations are only used in `nnnInit` and one-pass `nnn` calls, where no state is maintained (no `nnnUpdate` calls).

4. *Digest state and private RSA key*, including mechanism.

   Used during digital signature generation, with incremental calls.

5. *HMAC states*, used during symmetric-key based signing and signature verification.

6. *Symmetric cipher states*, used for symmetric-key encryption and decryption.

With the exception of mechanism+SPKI, all above states are encrypted and integrity-protected (p. 9). Composite states insert their own types to the Clic type system (`XCP_T_...` constants). Objects are serialized to single wrapped blobs, as described elsewhere. Clear objects are MAC-protected.

Certain entries of the table are crossed out with dashes. These combinations are not selected by PKCS#11 mechanisms, as they have no cryptographic meaning (i.e., RSA `Decrypt` with hashed private-key mechanisms).

Calls without intermediate storage (such as `SignSingle`) accept state inputs that are accepted by the corresponding `...Init` call. As an example, `EncryptSingle` accepts MACed SPKIs, while `VerifySingle` accepts raw SPKIs as well.

Note that `EncryptInit` and `VerifyInit` are different in one important detail: `EncryptInit` requires an SPKI of an already imported key, while `VerifyInit` works with any SPKI. Therefore, `EncryptInit` requires a MACed SPKI, while `VerifyInit` accepts SPKIs both with a MAC or "without MAC" **[WM]**.

Function	**mechanism +SPKI (in clear)**	**mechanism +state +SPKI**	**mechanism +priv. key**	**mechanism +state +priv. key**	**HMAC state**	**Symmetric cipher**
`EncryptInit`	✓	—				✓
`Encrypt`	✓	—				✓
`EncryptUpdate`		—				✓
`EncryptFinal`		—				✓
`DecryptInit`			✓	—		✓
`Decrypt`			✓	—		✓
`DecryptUpdate`				—		✓
`DecryptFinal`				—		✓
`SignInit`			✓	✓	✓	
`Sign`			✓	✓	✓	
`SignUpdate`				✓	✓	
`SignFinal`				✓	✓	
`VerifyInit`	✓**[WM]**	✓			✓	
`Verify`	✓	✓			✓	
`VerifyUpdate`		✓			✓	
`VerifyFinal`		✓			✓	

Table 1: State types vs. functions

5 Message encapsulation

Traffic between host and backend formatted in a simple TLV (tag, length, value) encoding. The set of patterns is an extremely limited form of ASN.1 encoding, representing all fields as OCTET STRINGs (tag `04`), and encapsulating them in a single SEQUENCE (tag `0x30`). The only variation is the number of fields, which is call-specific.

We assume that a reliable pipe is provided between host and backend code, there are no further requirements. *Transport may need to add further headers and encapsulation when transporting request/response data. With the exception of standard in-band headers used in IBM mainframes, such transport additions are generally not covered in this document,* and we assume EP11 code is augmented to deal with them transparently. (In practice, host libraries provided as part of EP11 development already includes several platform-transport variants as build-time configuration options.)

There are two kinds of parameters passed within messages, integers and buffers. Both integers and buffers are encoded as OCTET STRINGs; decoding must differentiate between the two. As the order and number of integer and buffer parameters is fixed for each handler function, this decoding is unambiguous. Fields are big-endian.

In addition to a command-specific number of "user" fields, a fixed number of system-level fields are inserted first. These fields are consumed by the transport layers, and are not visible to dispatch functions themselves. If system fields are missing or inconsistent, transport reports an error, and rejects the given message.

For requests, the main dispatcher verifies that the number of fields is consistent with the function code. Internal dispatch functions must verify the consistency of individual fields.

Responses are checked for consistency by transport. If the returned ASN.1 structure is inconsistent with its embedded function code, or the function code does not match that of the request, host code will stop parsing. Host functions are responsible for parsing individual fields, once basic consistency has been verified.

Error returns are regular module-to-host TLV packages, but only with system fields, and a single return value. This is a special case, where the returned structure is not checked against the corresponding field count. Certain functions do not return buffers, just values (such as `C_Verify`); those are flagged as having zero return buffers.

Optional or empty fields, possible for certain commands, are not removed from transfers. They are sent as empty fields, i.e., OCTET STRINGs with zero length—this encoding is valid, but unusual, although BER-parsing tools may legitimately flag it with a warning. The alternative, changing the number of fields, would make parsing ambiguous. Individual functions must verify that the structure of fields is what's expected; an error is returned otherwise.

Table 6 summarizes parameters of PKCS#11-derived EP11 functions, including parameter count and names. Parameters in parentheses are optional, parsing functions must detect their presence and act accordingly. Similarly, Table 7 describes non-PKCS#11 functions. (Note that both tables are automatically generated from EP11 source code.)

5.1 Transport message structure

Transfers are designed for single-pass processing: messages are self-contained. All required data is encapsulated in a single message, no further host interaction is needed once a request is received. Module-to-host responses are similarly self-contained, requiring no further interaction with the module.

Host code is responsible for routing returned values. Return values and user-visible results are returned or copied to the caller. Internal objects, such as opaque key blocks, are retained in by the EP11-aware host library.

The same transport structure is applicable to the real HSM backend, or software simulations (socket, DLL, or standalone code). The only difference is in transport mechanism. As long as the underlying transport passes opaque buffers reliably, the interface protocol is unchanged (in fact, it stays binary compatible if the backing CLiC versions are).

Transport implementation details are described in the shared header file `transport.h`. This file, intended for inclusion in both HSM and host (EP11) code, contains the table of parameter counts, ASN.1 formats, symbolic function identifiers, and all other low-level details. (The header file is also used to generate the summary tables in this document.)

6 Administration

Administrative commands are below the PKCS#11 level, and provide an almost orthogonal set of services. Not directly represented as PKCS#11 commands, administration involves the following groups of functionality:

1. *Import transport wrapping keys* in whole or parts
2. *Export transport wrapping keys* in whole or parts
3. *Reencrypt key material* between different transport wrapping keys (within the same domain)
4. Manage administrators
5. Change administrator signature (and revocation signature) thresholds
6. Change protection/operating mode of module (to non-FIPS, for example)
7. Disable features (such as key import or export)

The above manipulation is grouped into management of certificates, attributes—integers and bitfields—and controlpoints (Table. 2). Generally, state-changing commands must be signed by the appropriate number of administrators, with the specific exceptions of commands requiring no signatures. (The latter require host cooperation, and are assumed to be restricted by policies. None of the zero-signature commands reveal card-resident secrets, but they may advance backend state in accordance with the card security assumptions.)

At first power-up, or after zeroization, in *"imprint mode"*, modules are activated without registered administrators, and they start accepting administrator identities—certificates—without signatures. This mode is terminated by an administrator command, and would be performed under controlled conditions, before allowing user access to new cards. Once the card has been "imprinted" with its administrators' identities, it may be controlled over end-to-end channels, and may be made accessible to users.

We rely on x509 certificates representing administrators. *No certificate(signature) validation happens when adding an administrator; certificates are only used as portable public key containers. We do not interpret PKI context of any certificates we encounter.* Since we only parse actual public keys, backends are immune to any "vulnerabilities" of certificate verification—i.e., any logical PKI-hierarchy problems which may manifest in attributes or the hierarchy itself, not within the public key itself [BJR+14, IX] [GIJ+12, 7] [ASVH13, 4]. Unlike PKI clients which unintentionally do not validate certificate attributes [FHM+12, 5.3, 4.3], we ignore non-publickey parts by design. (Note that certificate attributes, even of certificates used by high-risk industries, are frequently assigned inconsistently [DKCC16].)

Public keys returned by the card are generally SPKIs, self-contained standard public-key formats; we do not construct proper certificates for these activities.

Administrator commands must be signed by a sufficient number of administrators, as described below. Changing the generic administration threshold, and a separate one for revocation, one can implement many feasible multi-administrator schemes.

We assume that administrators will disable unneeded capabilities depending on deployment before establishing transport wrapping keys. The backend does not enforce any policies on order of management, other than verifying the then-current number of signatures on commands.

6.1 Administrator setup

When a backend is first initialized, it starts in "imprint mode", without transportkey or administrator. As the first administrative actions, the card must be populated administrator certificates, which will be accepted without authentication (Fig. 7). We essentially instantiate "trust on first use" [WAP08, 1], and rely on externally verifiable auditing backed by module-resident signing keys (unlike online services [WAP08, 3.3], we provide queries to establish the provenance of keys).

As soon as a sufficient number of administrators is present, and a special signed request is submitted, the card leaves *imprint mode*. Imprint mode is no longer entered, unless the card (or domain) is zeroized, or a recoverable file failure removes administrators from it; in both cases, the affected unit/s revert to imprint mode.

If the card has administrators, but they may not be restored from persistent storage, we treat this as a failure and purge all persistent administrator and key storage state. *Note that administrators are higher priority in this case; if persistent administrators are corrupt, the keyfile must also be invalidated (erased). In the other direction, when key restoration fails, administrators need not be removed (but the module will start without a key, which must be generated or restored from outside).* Fig. 6 shows the relative hierarchy of removing/recreating module files after file corruption or other failure.

See also: Imprint mode

6.2 Administrator revocation

We provide a service to remove administrators, as part of regular administrator rollover. Revocation may not reduce the number of administrators to below the current signature threshold (in which case, the card would become impossible to administer).

Note that when active administrator private keys are lost, the backend may become unmanageable even without revocation. The transaction set is still consistent, card integrity is not compromised, but the card will no longer accept signed administrator commands (see the "Unmanageable" state in Fig. 7). Users are encouraged to supply additional, spare administrators, to reduce the chance of hitting this special case.

As a special case of revocation, we provide services to replace existing administrators. This may be performed as an atomic action, without changing thresholds, and without even transiently changing the number of active administrators.

6.3 Administrator authentication

Administrator commands are authenticated based on public-key cryptography, through public keys. An administrator identity is proven through the capability to sign with a private key corresponding to the administrator public key.

Administrator public keys are supplied during administrator login, through X.509 certificates, which are only used to retrieve public keys. *Other parameters of certificates, including signer identity and usage restrictions, are ignored.* Since there is no way of establishing external trust at this level, and some signing devices may have self-signed certificates, verifying signer identity is not considered feasible.

To provide a verifiable trust base, stored administrator keys are available through a dedicated query. One may obtain a list of certificate hashes, or actual certificates individually. This compensates the lack of discrimination based on CA signer; it is always possible to verify who is controlling a module.

Note that administrator authentication is not related to PKCS#11 services. Administrator identities are established before PKCS#11 commands are issued.

6.4 Administrative commands

Administrator commands must be authenticated, by administrators signing the command payload. The proposed method to authenticate is through PKCS#7 `SignedData` messages, possibly involving multiple signers (each administrator generates a single signature). Administrators can build such multi-signed messages easily, since the process can be partitioned into generating individual signatures (only RSA signature, no additional formatting), which could be performed on any signing device. A TKE workstation, smartcards, or even softtokens could trivially generate each administrator's signature. A trivial amount of wrapping code, as demonstrated by the existing Clic-based prototype, can combine individual signatures into a multi-signed PKCS#7 message.

There are several advantages of using multi-signer PKCS#7 messages for command authentication:

- The scheme works for an *arbitrary number of administrators* (including $N = 1$), *without modification*.
- No intermediate administration state—i.e., "partial commands" or similar state machines—need to be maintained inside the module (administrator code). *PKCS#7 verification is atomic, regardless of the number of signers.*

  State needs to be maintained by the external administrative process, but that complexity may be hidden trivially, as shown below. The result is moderate overhead on administrator hosts, no additional complexity inside the module. Specifically, since we do not expose a multi-stage state machine through chained transactions, host-induced attacks targeting the control flow of state machines are preempted [BBDL+15, II.]

  Operating only on reassembled, full commands allow us to prevent attacks on state sequences, such as combining partial transactions in insecure sequences, or similar abuse attempting to disrupt crypto-related state machines [BDLF+14, V] [BA01, 3.1] [Zol11, 4] [BBDL+15, IV.]. Obviously, *our backend only exports the problem to host (administrators), but this radically simplifies the backend implementation*—i.e., the only security-critical component.
- Host (administrator) overhead is limited to handling a single binary buffer (passing it through the list of administrators).

  With a suitable support library—see below—the only required administrator capability is to generate a single signature, and to pass the result reliably to the next administrator. Digital signatures make error-detection easy even during the incremental signature collection phase.
- Since PKCS#7 is a standard format, administrative messages could be constructed by existing tools.

Note that while we use individual PKCS#7 structures such as `signerInfos`, we repackage them into a proprietary transport package. However, we have implemented our transport with standard PKCS#7-packaging code as a technology demonstration, so we simply refer to PKCS#7 transport (primitives) in our documents.

A prototype implementation, based on Clic, is available. It can be trivially extended to handle non-Clic signers (such as signing devices), or replaced by equivalent utilities for platforms where Clic is not available. The process is straightforward:

1. Generate an empty PKCS#7 `SignedData` envelope with the given command payload, but without any signers.

   Such a "zero-signer" PKCS#7 message is technically legal, but obviously not very useful. It is not used on its own (lacking signatures, the module would reject it), but it's convenient to be able to separate PKCS#7 envelope creation and signing.
2. Each administrator uses an accessor function to parse and inspect the signed command (`admin_signature_payload`).
3. If the command is recognized as valid, generate an RSA signature through any signing device. Append the signature and the signer's certificate (`append_admin_signature`).
4. Repeat the previous two steps for all administrators.

The PKCS#7 payload obviously must use the certificates that are deposited to the module.

6.5 Administration lifecycle

Administration is divided into two distinct stages, initial administrator setup—imprinting—and operational administrator commands, both for card-level and domain-level administration (Fig. 7). Initial setup is limited to queries, and administrator management; setup must be terminated before functional requests are submitted to the module. Once all administrators have been registered, the module is ready to authenticate administrator commands, and may start receiving functional requests. At this point, all administrator commands become available, and every administrator action must be authenticated.

Note that in development builds, we may support thresholds of zero—all commands will be accepted without signatures. This is not possible in production code.

Registered certificates may be queried as lists of public key hashes—SKIs, SubjectKeyInfo's—or individual certificates (which return ASN.1/BER form). Certificates provide the public keys which are used to verify administrator signatures; other than providing a standard form of public-key storage, other certificate properties—such as expiration dates—are not verified. The lack of expiration verification mirrors PKCS#11 [PKC04, 10.6.2]; we apply the same rationale.

Administrative traffic is intentionally stateful, as opposed to the rest of functionality, to prevent replay of administrative traffic. *Administration state tracking* combines salt-like quantities and transaction counters to keep a strict order of administrator commands and prevent replay:

- *Administrator transaction counter*, a "sufficiently long" counter that starts at zero during installation, and is retained across module restarts. In order for a signed administrator command to succeed, the request must include a larger value of the same size. Once accepted, the counter is advanced to the provided value.

  State-changing commands which are accepted without signatures increment the transaction counter, ignoring any user-supplied value.

  There are separate transaction counters for the entire module, and for each domain.
- *Module identifier*, generated fresh upon every module zeroization or during first startup. Prevents replay of administrative traffic to the same module, if after zeroization it is populated by the same administrators as in any previous instance.
- *Domain instance identifier,* similar to the module instance identifier, restricting replay across the same domain.

Instance identifiers are populated with random data during first access, are reset during zeroization, and are retained across backend restarts.

We do not support transaction counter wrap-around; administration becomes impossible once the counter has advanced to its largest possible value. Note that counter size is selected to be infeasible to exhaust under reasonable circumstances.

Initial setup, typically performed by drivers performed before activating the card, allows registering initial admins (*"imprint mode"*). No authentication is performed during this stage; the only available commands perform administrator management.

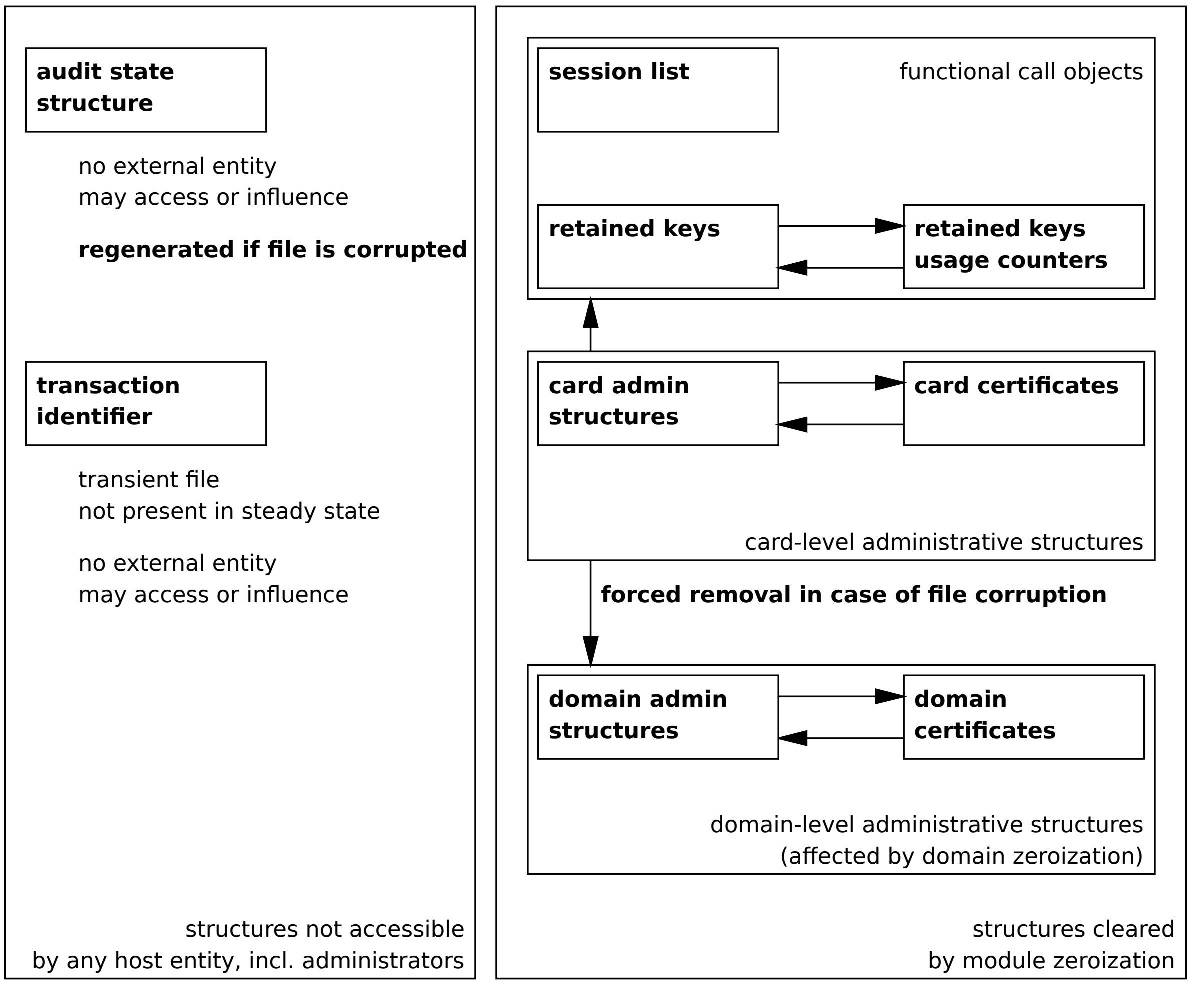
audit state structure
no external entity may access or influence
regenerated if file is corrupted
transaction identifier
transient file not present in steady state
no external entity may access or influence
structures not accessible by any host entity, incl. administrators
session list
functional call objects
retained keys
retained keys usage counters
card admin structures
card certificates
card-level administrative structures
forced removal in case of file corruption
domain admin structures
domain certificates
domain-level administrative structures (affected by domain zeroization)
structures cleared by module zeroization

Figure 6: List and interactions of backend-resident files

At the end of administrator lifecycle, *administrator revocation* may be initialized by the administrator, or other administrators. If an administrator removes his own certificate, no countersignatures are required, regardless of active administrator signature threshold. If other admins are removing a certificate, their signature must be over the—revocation—threshold.

If the number of administrators reaches N, the current threshold, the last remaining administrator may not revoke further ones.

6.5.1 Administrator signature counts

Administrator signatures are applied in the internal two-level hierarchy: card-level administrators may override domain-level ones. When counting signatures, the following rules apply:

1. All present signatures (SignerInfo's, see the wire section for details) must apply to the same payload, and all must be valid.

   Only administrator signatures from authorized keys are counted.
2. Card-level commands must be signed by only card-level administrators.
3. When counting domain-level signatures, a card-level signature also counts as one.
4. Only one signature per administrator key (SKI) is accepted per command.
5. The total number of signatures must be over the generic threshold, or the applicable revocation one (administrator revocation only). The threshold is unambiguously the card-level one, or that in the targeted domain, depending on command.

It is generally assumed that card-level administrator keys are used infrequently, and most administration uses lower-assurance, domain-level keys. In practice, we assume that domain and card-level signatures will not be mixed.

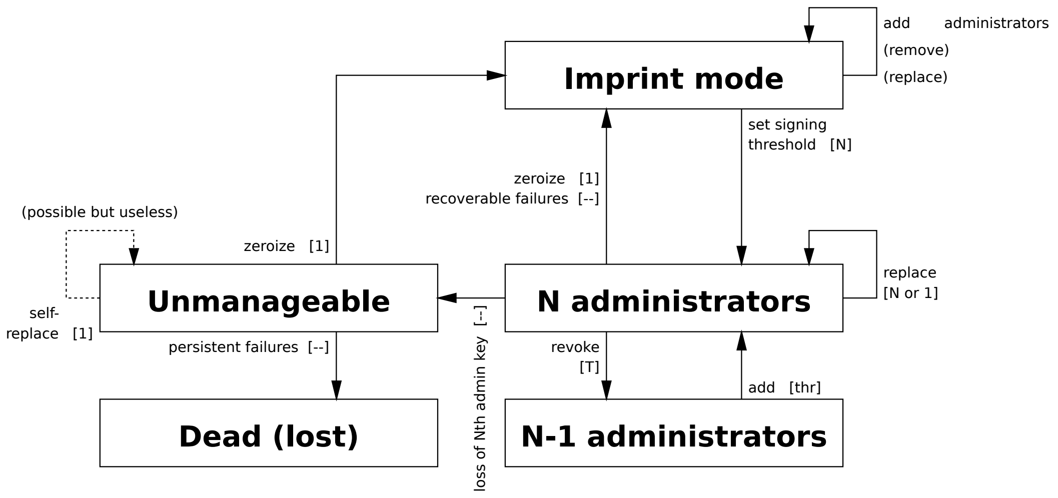

Figure 7: Administrative lifecycle

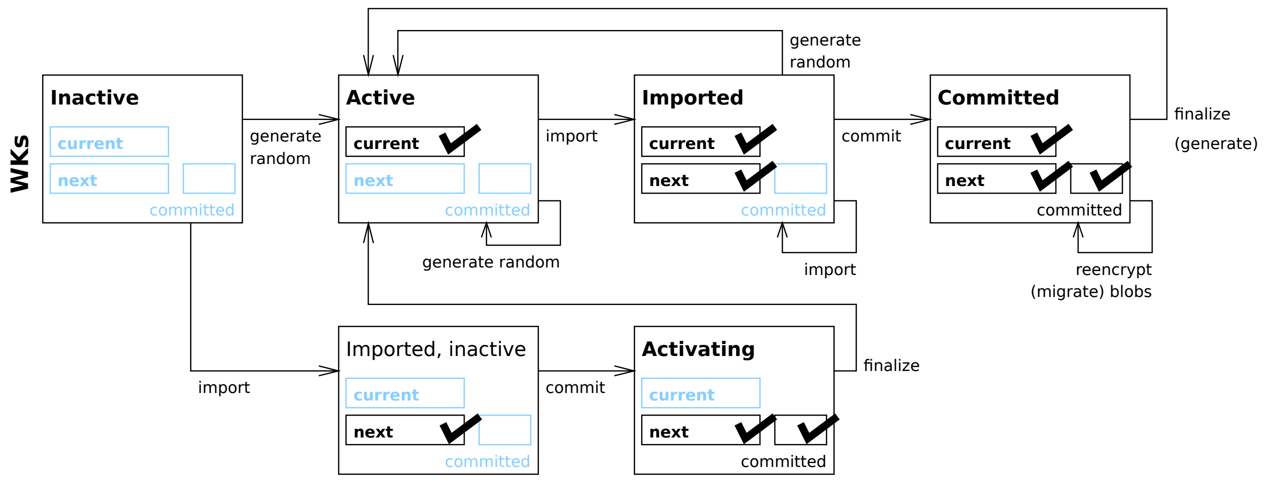

Figure 8: WK lifecycle

6.5.2 Imprint mode

Administration starts without predefined certificates or CAs in the factory state, or after zeroization. In this state, *"imprint mode,"* the initial population of administrator certs is deposited to the card/domain, initial thresholds are established, then imprint mode is terminated. Our assumption of initially connecting entities allowed to take control, then subsequent attacks protected by end-to-end authentication, mirror security expectations of distributed applications under comparable security goals [WAP08, 1] [EPS15, 1].

The following special rules apply to imprint mode:

1. Commands are accepted without signatures. The first successful command increasing the signature threshold from zero terminates imprint mode: all subsequent commands must be signed as described in Table 2.
2. Only the following commands are available in imprint mode:
   - Login administrator
   - Logout administrator
   - Replace administrator
   - Set attribute/s
   - Set FCV (card-level imprint only)
   - Zeroize

   Queries may be issued without restrictions. Note that certain queries—such as those involving WKs—may not produce meaningful results, but they are not otherwise restricted.
3. Administrator-management commands are accepted without signatures. Setting attributes requires one or more signatures when setting the signature threshold. Setting attributes other than the signature threshold—such as the revoke threshold—is allowed without signatures.
4. Once a threshold has been raised from non-zero, it may not be lowered back to zero.
5. Imprint mode may only be terminated if both signature and revoke threshold has been increased to non-zero. It is not relevant whether the revocation threshold has been increased first, or both new thresholds are specified in a single "set attribute" command.
6. *The command to terminate imprint mode—increasing the signing threshold from zero—may be signed by a single signer,* even if "set attribute" is otherwise an N-signed command (see Table 2).

   Note that when the command is issued, the then-current threshold would be zero, we just interpret it as 1 in this case.
7. The command to terminate imprint mode, when increasing threshold from zero to N, must be signed by at least N currently registered administrators.

Any additional checks apply to imprint mode unchanged. As the most relevant example, none of the thresholds may be increased to N if the targeted card/domain has less than N current administrators.

6.6 Control points (CPs)

Control points restrict specific capabilities within a domain. They are represented in an array of individual CP bits (CPBs), with a set bit enabling the capability. Future additions will preserve this positive-active logic, therefore zero-extending a set of CPs from a previous release to a more recent version will remain a safe operation.

The following CPs influence core infrastructure:

- Allow addition/activation of CPBs (`ADD_CPBS`). A separate CPB allows removal of CPBs (`RESTRICT_CPBS`).
  To make the setup read-only, deactivate both of these CPBs. Removing only addition, but not deletion, turns the setup into a form that may be further restricted, but missing capabilities may no longer be activated.
- Allow the backend to save blobs as semi-retained keys (`RETAINKEYS`), see (2.1).
  Note that this setting does not influence key caching, which may not be externally controlled, and is host-opaque.
- Allow modification of object attributes (`MODIFY_OBJECTS`). Removing this attribute makes all objects read-only.
  Note that currently, only Boolean attributes may be modified.
- Allow mixing of external seed to the backend, if it is supported (`RNG_SEED`).
- Allow generating asymmetric keys without selftests—private/public key operation consistency checks—upon asymmetric key(pair) generation (`SKIP_KEYTESTS`).

The following control points influence groups of functionality:

- Allow signing with asymmetric (private) keys, symmetric keys (`SIGN_ASYMM`, `SIGN_SYMM`) or verification with symmetric keys (`SIGVERIFY_SYMM`).
  Note that one can not restrict signature verification with public keys, which are available to the host.
- Allow encryption of data with with symmetric keys (`ENCRYPT_SYMM`), decryption with symmetric or asymmetric ones (`DECRYPT_SYMM`, `DECRYPT_ASYMM`).
  Encryption with public keys can not be prevented, therefore there is no such CPB. (Note that *wrapping* keys is separately controlled.)
- Allow generation of symmetric or asymmetric keys (`KEYGEN_SYMM`, `KEYGEN_ASYMM`) or key derivation creating new symmetric keys `DERIVE`.
- Allow wrapping keys with symmetric or asymmetric keys (`WRAP_SYMM`, `WRAP_ASYMM`).
- Allow unwrapping keys with symmetric or asymmetric keys (`UNWRAP_SYMM`, `UNWRAP_ASYMM`).
- Allow cryptographic strength windows: below 80 bits, 80 to below-112, 112 to below-127, 128 to below-192, 192 to below-256, or 256-bit (`KEYSZ_BELOW80BIT`, `KEYSZ_80BIT`, `KEYSZ_112BIT`, `KEYSZ_128BIT` `KEYSZ_192BIT`, `KEYSZ_256BIT`).
- Allow algorithms not allowed by NIST or BSI rules of a specific date (`ALG_NFIPS2009`, `ALG_NBSI2009`, `ALG_NFIPS2011`, `ALG_NBSI2011`)
  These CPBs control entire sets of algorithms. They may be further restricted.

The following CPBs are specific to algorithms or other PKCS#11-level functionality:

- Allow keywrapping without attribute-binding (`NON_ATTRBOUND`). This CPB must be set for standard PKCS#11 key transport, which uses key forms without attributes.
- Allow raw—unpadded—RSA operations (`ALG_RAW_RSA`)
- Allow HMAC keys below the minimum FIPS-198 keysize (half of state size) (`KEYSZ_HMAC_ANY`)
- Allow RSA public keys below $2^{16}+1$ (`KEYSZ_RSA65536`). This restriction corresponds to the lower limit introduced by FIPS 186–3 (at the end of 2010).
- Allow functional use of RSA, DSA, Diffie-Hellman or elliptic curves (`ALG_RSA`, `ALG_DSA`, `ALG_DH`, `ALG_EC`)
- Allow EC operations over NIST or Brainpool (E.U.) curves (`ALG_EC_NISTCRV`, `ALG_EC_BPOOLCRV`).
- Allow non-administrators to set objects' `CKA_TRUSTED` attribute, which in turn allows export of keys restricted to transport with trusted keys (`USER_SET_TRUSTED`).
  Note that non-administrator-controlled `TRUSTED` attributes are inherently unsafe, and need proper privilege separation on the host.
- Allow creation/use of keys which can un/wrap and en/decrypt simultaneously (`WRAP_CRYPT_KEYS`). Similar restrictions are possible to prevent sign/verify and and en/decrypt `SIGN_CRYPT_KEYS`) or un/wrap and sign/verify (`WRAP_SIGN_KEYS`). The CPs apply to all combinations of symmetric and asymmetric keys.
  These restrictions, when enforced, prevent compromise of key material through misuse of blob attributes, such as mixing encrypted *keys* and *data* [BCFS10, Clu03]. Adding such restrictions prevents the entire category of such attacks.

Control point restrictions are cumulative: all applicable CPs must be enabled for an operation to succeed (see section 2.6). Error reports are not—currently—specific about which CP has caused the request to be rejected. Since CP verification is essentially searching for an intersection of permissions' and requirements' bitvectors, once CP count gets sufficiently high, bitvector-aggregation techniques used by firewalls are directly applicable [LLS03, III.B].

While CP restrictions are basically dynamic, backends MAY reject certain configurations completely. This would prevent, for example, certain CPs from being ever activated, which is backend-dependent. To indicate such possibility, in addition to the regular "policy prevents operation" return value, we reserved another vendor-extension return value to indicate "policy prevents operation,and this backend will never allow reconfiguration to allow it". Host code MAY choose to log these conditions separately.

We currently do not support Control point "profiles", symbolic references to full collections of control points. Host applications therefore must specify the full set of CPBs to set modes. Since profiles would be unambiguously distinguishable from sets of CPBs, profile support may be transparently added in the future.

6.7 Operational modes (compliance)

The backend supports a number of Boolean *operational modes*, predefined lists of control points which correspond to specific—security—standards. The specific combinations they represent is detected, whenever the combination of control points fulfills the requirements. *Mode settings are controlled indirectly: attributes reporting compliance are read-only, updated once CPs have been set.*

Operational modes correspond to compliance with the supported standards. By including expected compliance bits within objects, we bind objects to specific sets of standards, and prevent their use in any other environment. Compliance effectively segments host-based keystores into mutually incompatible subsets, where objects of different compliance setups are prevented from functional interaction.

Operational modes are reported hierarchically: card-level mode bits show the logical AND of all domains with WKs (domains without WKs may not service functional requests, therefore are ignored for API-level standard compliance). Since card-level mode state is a conservative overview of all domains' modes, it serves as a single query when reporting card-level state (i.e., can be used to provide a single "FIPS mode" or "BSI mode" indicator for a card with multiple active domains).

Domain and card modes are recalculated after the following administrative actions:

1. Modification of control points
2. Domain "activation," when a domain WK gets activated (i.e., the domain becomes capable of accepting functional requests)
3. Domain deactivation or zeroization, if it turns a previously functional domain inactive

We ignore compliance settings for domains without WKs as they may not process functional requests. Our definition of compliance modes—see the security policy—states that we only enforce their restrictions for functional requests.

We support multiple revisions of *FIPS-140* and *BSI (HSM) protection profile* compliant modes. These compliance settings are usually comparable, and domains may be in a mode which simultaneously compliant with more of them. Simultaneously activated modes support only the intersection of algorithm lists, and observe the union of usage restrictions.

We foresee the list of these mode-selecting options to grow in the future, such as with future revisions of the underlying standards. As with control points, we will append future mode settings to the list without changing existing ones.

6.7.1 Compliance mode inheritance rules

Compliance mode settings are enforced for blobs and MACed SPKIs, including non-key objects embedded within blobs (such as encrypted state structures of incremental operations). Enforcement of compliance follows the following lifecycle rules:

1. Compliance mode may be supplied as an attribute during key generation, unwrapping, or derivation. Attributes are supplied as one or more integer attributes containing a bitmask of compliance bits (see `CKA_IBM_STD_COMPLIANCE...`).
2. If no compliance mode is supplied during key generation or unwrapping, the current setup of the domain is inherited.
3. Newly generated keypairs' compliance settings are supplied separately, each within the corresponding public/private-key attributes.
4. If a key is derived from another one, and no compliance mode is provided, the derived key inherits the compliance mode of the originating blob.
5. If keypair generation provides private attributes with an explicitly supplied compliance mode, and the public key does not provide one, the public key mode is inherited from those of the private key.

6. Object creation may succeed, even if the resulting objects' compliance mode differs from the current setup of the generating domain, if key generation methods are permitted by both the current and targeted compliance modes (see exception below).
7. *Object creation may be rejected,* reporting a policy failure, if the selected key generation method would not be acceptable for the requested compliance mode, even if it is accepted by the current domain setup.

   Note that currently, we do not support algorithms with such conditions, therefore no such rejections are performed. However, we list this option as a future possibility, if algorithms and compliance modes are added in the future.
8. *For functional calls, objects are rejected if their compliance mode does not match that of the domain it is submitted to.*

   Queries or modification—i.e., non-functional access to object contents—of object compliance (see below) are possible even if object and domain compliance differ.
9. Attribute-bound objects transport their compliance settings, as part of the full set of attributes they include.
10. Objects derived from blobs, such as incremental operation states, inherit their compliance mode from the originating object.

    Note that this differs from *key derivation*, where a new set of attributes may be supplied: when initializing a state object from a blob, one may not supply attributes of the newly created state object.

Compliance mode bits are stored in two locations, within blob attributes and a separate copy—a dedicated field—in the blob header. A newly created object will contain identical copies, but the two fields may subsequently diverge.

6.7.2 Compliance mode modification

When an object compliance is modified, only the dedicated header field is updated. The original compliance field, within object attributes, is never updated, and therefore it represents "object compliance history." Currently, original compliance is not used functionally, but it may be in the future—such as evaluating whether an object has been created under certain conditions (which may be inferred from its original compliance settings).

Compliance mode is updated through calls to `SetAttributeValue()`, which is allowed even if current domain and object compliance differ (6.7.1). Modification rules are the following:

1. *Updates may add, but not remove, compliance bits from an object.* New compliance is supplied as a change to the corresponding attribute (bitfield).

   Preventing removal of updates prevents objects from being migrated to more relaxed environment, but allows adding further restrictions to an existing object. (Since objects also retain their original compliance, more granular cross-checks may be added in the future.)
2. The targeted new compliance need not match the current mode of the targeted domain. Objects so updated become functionally inactive until the domain mode is subsequently changed. Such mismatch would be intentional, for example, when migrating a keystore to a preannounced, stricter setup due to a future domain mode change.
3. Specifying all-zeroes as the "new compliance" bitmask update the object to the then-current mode of the targeted domain. (If the current mode is a subset of existing object compliance, the update is rejected.)

   Note that all objects contain at least one compliance bit, therefore all-zeroes is not a valid "new compliance" combination, and the special case is unambiguous.

The effective compliance mode is the stricter subset of the two compliance bit fields; by construction, the blob-embedded version is read-only (it may not be modified after blob creation).

Currently, all domains operate in a mode compliant with FIPS-140 in the setup corresponding to settings allowed 2009. Future default modes may add further restrictions by default.

6.7.3 Irreversible attribute settings

The most critical set of card/module attributes are controlled through a two-step process, which allows modification of the attribute, and provides settings ("meta-attribute") to prevent subsequent modification of the attribute itself. For each such critical attribute, a corresponding "CHANGE-..." attribute is also available, initially in an enabled state. Removal of the meta-attribute permission prevents reactivation of the meta-attribute or any further change to the controlled attribute.

The capability to prevent further modification makes it possible to create setups for specific uses, and ensure that no further administrator action may subsequently relax the security setup. *Especially in high-security scenarios, additional assurance may be derived from the fact that even administrators are unable to change some of the setup,* such as to disable restrictions which are currently enabled. A typical application would be to prohibit key or state export, then remove the capability to reactivate it: any user keys subsequently loaded into a backend in such a state would be de facto non-extractable until the module is reset to factory state (through firmware management, outside EP11 control).

Since meta-attributes may not be reactivated, the irreversibility of their removal allows others to partially distrust even administrators. *Administrators may therefore demonstrate full commitment by removing their own ability to control certain settings, when activating—without the capability to subsequently deactivate—security controls* [Kub64].

The list of attributes which are mirrored to a meta-attribute allowing modification is currently:

1. Allow state export
2. Allow state import
3. Allow WK export
4. Allow WK import
5. Allow WK transport in one piece (i.e., without use of multiple keyparts)
6. Allow use of randomly generated WKs
7. Allow changing signature or revocation thresholds (separately)
8. Enable single-signed administration (i.e., setting thresholds to 1)
9. Enable single-signed CP changes
10. Enable single-signed zeroization changes

Settings allowing change of the above attributes are initialized as all-permissive. They may be removed, but may not be added back. See 8.5.1 for examples setups which are controlled through elements of the meta-controlled attribute list.

Host tools are recommended to warn users about making irreversible changes. Note that all these changes persist until zeroization, and are therefore irreversible only in the current instance of the targeted card/domain.

7 Administration message formats

Administrative requests are packaged in a single, sub-typed administrative structure. The single administrative interface manages all state through the single, non-PKCS#11 function (`m_admin()`).

Administrative blocks combine the following information into SEQUENCEs (see `xcpAdminBlk`):

1. administrative function identification
2. target domain (if applicable, otherwise fixed 0 for card-level blocks)
3. module identification
4. new transaction counter (module or domain-level), mandatory empty for queries
5. command-specific payload, possibly empty

Module identification combines the module serial number, and an "instance identifier," a random value generated upon first run (with a negligible chance of repeating). We combine these values to distinguish different instances of the same firmware on the same module, preventing replay between different runs of the same firmware.

On modules without persistent serial numbers, such as soft-HSMs (software-backed, socket-attached backends), module identification may be constructed completely randomly.

The size of instance identifiers and randomly generated components makes collisions' frequencies negligible, but backends do not check explicitly for repetition. *We intentionally tolerate a negligible chance of collision, assuming procedural controls will detect if a card or domain is zeroized too frequently—which is the only way of regenerating instance IDs.*

The new *transaction counter* must exceed the current one (within the target domain, or the card-global one, depending on the command target). The field is sufficiently large to allow large increments; we do not support counter wrapping. If the counter ever reaches its—infeasibly large—limit, no more commands may be issued. Counters are reinitialized to zero upon firmware reload, or upon domain zeroization.

Administrative blocks, when constructed, are further embedded into a request to the administrative call, as its payload field (see the `xcpAdminReq` definition in the wire section). Administrators sign the entire administrative block as a constructed SEQUENCE, and their signatures are collected into a separate parameter, which must all apply to the same "payload" (i.e., the administrative block). The number of required signatures depends on block type—see the threshold (Thr.) columns in the command tables—while queries are all unsigned.

Card-level administrative traffic must be targeted to a domain *at a transport level,* but requires an all-zero domain field within its command request block. We depend on the capability to unambiguously distinguish card-level requests from domains based on the function identifier. Domain field within the request must be consistent with transport-level domain fields—such as within the CPRB header for domain-level actions.

Queries may always be issued, even during imprinting.

Individual administrator services accept the following input, and return the following output on success:

Login card administrator, Login domain administrator supply certificate to add; returns updated list of SKIs in the modified set (card or domain).

Returns `CKR_USER_ALREADY_LOGGED_IN`, failing, if the provided certificate is already registered. When attempting an unsigned—imprinting—addition, `CKR_SESSION_CLOSED` is returned if imprinting mode has been terminated. If the number of administrators would rise to over the supported maximum (`XCP_MAX_ADMINS`) in the targeted unit–card or domain–the return value is `CKR_USER_TOO_MANY_TYPES`.

Unlike most administrator commands, Login may advance the system through more than two states atomically (section 2.5.1). This distinction is not observable, unless execution of the command is interrupted, which may require administrators to re-submit the same request with an updated transaction counter.

Logout card administrator, Logout domain administrator supply a list of SKIs of certificate(s) to remove; returns updated list of SKIs in the modified set (card or domain). Note that the number of required signers differs from generic removal of any certificate (N signatures needed) and an administrator removing himself (a single self-signature is sufficient).

Removal may not reduce the number of administrators below the current threshold (N). Note that reducing the number to exactly N is technically possible, but discouraged: losing any key due to malice or malfunction leaves the backend—or domain—in a state where it may be no longer managed.

Function	*Payload (request/response)*	*Thr.*	*Notes*
Administrator management			
Login card administrator	certificate	N/0	domain is zero; not signed during imprinting
	updated list of administrator SKIs		
Login domain administrator	certificate	N/0	not signed during imprinting
	updated list of administrator SKIs		
Logout card administrator	SKI of administrator	N/1/0	domain is zero; self-logout requires single signature; not signed during imprinting
	updated list of administrator SKIs		
Logout domain administrator	SKI of administrator	N/1/0	self-logout requires single signature; not signed during imprinting
	updated list of administrator SKIs		
Replace card administrator	SKI of previous administrator; certificate of new one	N/1/0	domain is zero; self-replacement needs single signature; not signed during imprinting
	updated list of administrator SKIs		
Replace domain administrator	SKI of previous administrator; certificate of new one	N/1/0	self-replacement needs single signature; not signed during imprinting
	updated list of administrator SKIs		
Key migration and management (domains only)			
Create random WK	N/A	1	
	verification pattern of new WK		
Import domain WK	list of parts, individually signed	0	domains only see reassembly rules; signatures checked independently
	list of VPs: committed WK, those of originating keyparts		special case for single-part WKs (see text)
Commit pending WK	verification pattern of pending WK	N	must have pending, reassembled, matching WK
	list of VPs: committed WK, those of originating keyparts		special case for single-part WKs (see text)
Finalize WK	verification pattern of pending WK	0	erases previous key; activates (pending) current key
	verification pattern of current WK		
Export domain WK	list of certificates (recipients)	N	domains only
	list of encrypted keyparts		
Reencrypt (WK transfer)	blob (current WK, active session)	0	domain must have current and pending WK
	blob (next WK, same session)		response is not signed
Clear WK	(empty)	1	domains only
	(empty)		
Clear pending WK	(empty)	1	domains only
	(empty)		
Importer public key management			
Generate importer key	key type	1	domain
	SPKI of new importer		
Generate module importer	key type	1	card command
	SPKI of new importer, with signature		signature concatenated without further formatting
State cloning (card-level commands only)			
Export state	(empty)	N	(see format)
	bytecount of serialized state and keyparts' file		
Import file (part)	file contents with file-part header	1	(see format)
	file-part header of written data		
Commit imported state	(empty)	N	full state and KPs must have been imported
	(empty)		
Remove cloning state	file-part header (optional)	1	erases all state files without payload; only designated file if present
	file-part header, or empty		original input payload
Administrative settings			
Set card attributes	attributes to change (packed)	N	domain is zero
	updated card attributes, full set (packed)		
Set domain attributes	attributes to change (packed)	N	
	updated domain attributes, full set (packed)		
Control point management			
Set control points	full CPB set	N/1	threshold attribute-dependent
	updated CPB set		
Add control points	full CPB set	N/1	ORs CPBs with current set; threshold attribute-dependent
	updated CPB set		
Remove control points	full CPB set	N/1	removes CPBs from current set; threshold attribute-dependent
	updated CPB set		

Table 2: Administrator commands (administrative actions)

Function	*Payload (request/response)*	*Thr.*	*Notes*
Infrastructure control			
Set clock	current date/time (UTC)	1	domain is zero; *card-level only*
	current (updated) date/time (UTC)		
Set FCV	new FCV	0	domain is zero; *card-level only*; accepted only once retained until card restart/zeroization
	public parts of FCV, packed		FCV intentionally not reported (is export-controlled)
Zeroization and other deletion			
Zeroize domain WK	(empty)	N/1/0	domain from encapsulation; single signed with system certificate; not signed during imprinting
	(empty)		
Multi-domain zeroize	domain mask	N/1/0	card command; single signed with system certificate; not signed during imprinting
	domain mask (updated)		
Zeroize card	(empty)	N/1/0	Application-level zeroize, does not affect OA; single signed with system certificate; not signed during imprinting
	(empty)		
Zeroize card (system)	(empty)	N/1/0	Zeroizes card, leaving any system certificate unchanged; single signed with system certificate; not signed during imprinting
	(empty)		
Remove (semi-) retained key	(S)RK identifier	1	Truncated identifier
	(S)RK list		updated list, after removal

Table 3: Administrator commands (infrastructure)

Query	*Payload (request/response)*	*Thr.*	*Notes*
List card administrators	SKI (specific cert) or empty (full list)	-	domain is zero; SKI is raw (no encapsulation)
	requested certificate, or list of SKIs		
List domain administrators	SKI (specific cert) or empty (full list)	-	SKI is raw (no encapsulation)
	requested certificate, or list of SKIs		
Query OA certificate/s	index (single) or empty (certificate count)	-	index is retroactive, 1: current, 2: previous. . .
	requested certificate (in native form)		
Query importer (public) key	N/A	-	for domains
	SPKI of current importer		
Query module importer	N/A	-	for module
	SPKI of current importer, incl. signature		
Query card control points	N/A	-	domain is zero
	CPB set		
Query domain control points	N/A	-	
	CPB set		
Query card attributes	domain-aggregate mask (optional)	-	domain is zero; see note
	current card attributes (packed)		
Query domain attributes	N/A	-	
	current domain attributes (packed)		
Query current WK	N/A	-	for domains only
	VP of currently active key		full verification pattern, not truncated
Query pending WK	N/A	-	for domains only
	VP of imported pending (next) key		full verification pattern, not truncated
Query WK origins	N/A	-	for domains only
	list of VPs: committed WK, those of originating keyparts		special case for single-part WKs (see text)
Query FCV	N/A	-	
	public parts of FCV, packed		not reporting full structure
Query retained keys	N/A	-	
	list of module-resident (S)RKs		truncated IDs only
Query cloning state (file)	file/part specification	-	card query
	file/part specification, updated bytecount field		see wire format
Query audit state	audit event specification, if any	-	card query
	audit record		see wire format

Table 4: Administrator queries

Returns CKR_USER_TYPE_INVALID, failing, if not all supplied certificates—those to replace—are already registered administrators. Note that this failure differs from what's returned if invalid signers authenticate the command.

See comments about atomicity under Login (above) and in section 2.5.1.

Replace card administrator, Replace domain administrator supply an SKI of certificate to replace, and the replacement certificate; returns list of SKIs in modified set (card or domain). Note that an administrator replacing himself requires only a self-signature, but replacing other administrators requires the usual consensus (N signatures).

Return values are those of "Logout" or "Login", with checks for Logout performed first.

See comments about atomicity under Login (above) and in section 2.5.1.

Generate importer key creates a new importer keypair for the target domain, with a caller-supplied type (see XCP_IMPRKEY_... constants).

Returns CKR_KEY_TYPE_INCONSISTENT if the provided keytype is not valid, or CKR_KEY_SIZE_RANGE if the backend does not support this keytype. Backends must support at least one of the possible importer types.

Generate module importer is functionally identical to *Generate importer key,* creating a module-level private key importer for state cloning (import). Note that due to historical compatibility reasons, module importers are returned with a directly concatenated signature; see importer queries for details.

Create random WK initializes a domain WK, or a pending WK with a new, internally generated uniform-random one. If there is no domain WK, a new one is generated. If there is a current WK present, but no pending one, a pending WK is generated and committed, allowing immediate WK migration, and subsequent finalization (Fig. 8).

If current and pending WKs are both present in the targeted domain, the command fails and returns CKR_OPERATION_ACTIVE.

Import domain WK accepts a set of encrypted keyparts (i.e., RecipientInfo's), all targeting the current importer, individually signed, wrapped in a single unsigned "import domain WK" envelope. It returns the verification pattern of the entire key, followed by verification patterns of all accepted keyparts, the latter in arbitrary order. KPHs must verify that their keypart's verification pattern is present in the response.

May return the following failures, in decreasing priority:

1. CKR_KEY_HANDLE_INVALID if no importer is present.
2. CKR_KEY_CHANGED if not all KPs target the same, current importer.

   Importers are implicitly targeted, within the command (i.e., KPs are encrypted for a particular, single-use importer). This targeting information is host-visible.
3. CKR_TEMPLATE_INCONSISTENT if keyparts do not correspond to the same reassembled WK. This can only happen with KPs exported from another module, or a setup where KPHs can verify the full key while holding only a part.

Keypart signers must all be administrators authorized for the target—module or domain—but need not be distinct: all parts may be signed by the same administrator. Dual control is enforced during the *commit* of keyparts, but not during actual import.

Domain targeting, transaction counter, and other auxiliary information of all embedded keyparts must match that of the encapsulating command block. (By implication, this provides some authentication on the full compound, even if it is not signed, as it must match targeting within its constituent, signed command blocks.)

An imported key is available, reassembled, but it is not yet active. The next logical command, *Commit WK*, serves this purpose (Fig. 8). Administrators must verify that the reported key VP—and its keyparts, if any—are consistent, before authorizing further use of WKs. Such explicit confirmation delineating commit stages between mutually suspicious parties based on public data can remove trusted intermediaries from multi-administrator management [BMC$^+$15, VIII.A.].

Upon success, the previously active importer private key is destroyed.

Commit WK transfers a reassembled, imported key from inactive to active state. It gets flagged as capable of reencrypting blobs from the current WK, but does yet not become the current WK. This action must precede "finalize."

Commit returns `CKR_OK` if called repeatedly, even after the first call has committed the pending WK. If attempting to commit a domain without a pending WK, `CKR_KEY_HANDLE_INVALID` is returned. Returns `CKR_KEY_CHANGED` if the WK verification pattern in the payload does not match the one being finalized—this includes malformed verification pattern.

Finalize WK activates a committed WK, erasing its previous one. The domain loses the capability to migrate keys to the current one. After completion, the previous WK has been completely eliminated, content encrypted by it becomes inaccessible.

Note that finalizing a WK is an exceptional, unsigned administrative command. It is under the complete administrative control of the host, and we require the same host administrative code to enforce update policy. Since the command only completes an already processed transaction, without exposing card-visible data, it does not endanger card secrets or confidentiality, even without authentication. *The transaction counter included in the incoming command block is ignored; success increments the counter.*

Returns `CKR_KEY_CHANGED` if the WK verification pattern in the command does not match the one being finalized—this includes malformed verification pattern. If attempting to finalize a domain without a current, and a committed pending WK, `CKR_KEY_HANDLE_INVALID` is returned.

Export domain WK takes a list of certificates of the intended KPHs ("recipients"), and returns a list of encrypted keyparts (i.e., PKCS#7 RecipientInfo's), and the verification pattern of the entire key. Encrypted keyparts include verification pattern inside the encrypted portion, allowing the KPH to verify integrity.

Clear WK erases the current WK in the targeted domain. Its purpose is to support immediate WK revocation, which "Create random WK" does not provide. Since this functionality is a subset of zeroization—which is also single-signed—this command is available even if attributes prevent creation of random WKs.

Reencrypt (WK transfer) transfers ownership of a blob controlled by a domain's current WK to the pending one, once the latter has been committed. It may be invoked with any blob—key or operation state—controlled by the current WK, if the blob's controlling session is also active. Under these conditions, the blob is reencrypted by the pending WK, and the WKID field is changed to reflect this. An updated blob is returned, with any attribute other than the WK—including blob size—remaining unchanged.

Similar to functional requests, this call may return `CKR_IBM_WKID_MISMATCH` if the current WK does not match that of the blob, or `CKR_IBM_BLOB_ERROR` if the blob is otherwise invalid.

Since reencryption requests are not signed—they do not change administrative state, only host-resident data—and they are assumed to be performed on large amounts of data, *reencrypt responses are not signed* and therefore feature an empty signature field. Since this command will need to be executed frequently when large keystores are migrated, and it is essentially a functional—non-administrative—one, not signing responses saves significant processing power (for signatures which would otherwise be ignored). *Note that audit records are generated for reencrypted objects.*

Export state creates a partially encrypted copy of all extractable module-internal state, storing it within the module. The symmetric encryption key encrypting sensitive portions is public-key-encrypted for a number of KPHs, whose certificates must be provided within request parameters (see the wire form for details).

The prerequisite for a successful export is a TLV-encoded structure specifying export parameters. Since this request is populated within the filesystem, the export command itself requires no direct input.

After a successful export state response, the module retains the exported state until it is restarted, or *Remove cloning state* is called. Exporting the state response is done through a sequence of *Query cloning state* requests, each returning a part of the module-resident state structure. (There is a transport-imposed limit on the size of file parts.)

In addition to the signature on responses, the exported structure is signed by the originating module. It also contains some information in its clear header about the export request which originally created it (see wire formats for details).

Note that retained keys and their attributes are not exported. No indication about the presence or absence of retained keys is included in cloning state structures.

The response payload contains packed 4-byte raw integers, total bytecount of the exported-generated files, in increasing file-identifier order. (The same information is available through size queries, obviously.)

Import file (part) deposits—parts of—persistent file into the targeted module. The first import-state request is used to report total bytecount; subsequent requests are only allowed to update parts of the allocated structure. Other than verifying the request signature, no integrity checks are performed on deposited parts.

Note that importing state parts is a purely administrative function, and does not require the cooperation of KPHs.

Internally, the module only maintains a single cloning state structure, and one file for storing KPs (see also: Fig. 13). Starting a new export or import state command discards any previously existing state before initializing a new one. (Mixing intermediate states of import and export operations is not practical, therefore this limitation should not impact regular export/import procedures.)

A single, global instance is selected for simplicity of implementation. Internal attributes unambiguously distinguish between import and export state, and invalid use of of state (such as activating an export structure) are rejected.

Files are unambiguously selected by file identifiers; see the wire section for a list of supported values. Note that file identifiers for Export and Import state are identical, since they both use serialized state and keyparts, even if keyparts are recoded during import (Fig. 13).

Commit imported state must be called when the full state structure has been called and reassembled, and an importer key has been generated (through *Generate module importer*). The commit request must include keyparts encrypted for the active module importer, identical to how *Import domain WK* transports a WK. The reassembled symmetric key is used to decrypt the cloning state, which is deposited within the module.

The imported state fully replaces the currently active setup. All administrative parameter, including administrators—even those authenticating the commit—are replaced by the set included in the cloned state.

After successful state commit, the module-internal state structure and the module importer key are removed.

Cloning state is retained only within transient module-internal storage. If the cloning procedure is interrupted by a module restart, the import or export procedure must be repeated. The time window where such restart may disrupt procedures may be minimized procedurally:

1. Exported state may be extracted immediately after cloning through queries—i.e., without active administrator intervention.
2. Importing (parts of the) cloned state is possible without KPH intervention, therefore may be repeated by administrators before prompting KPHs to encrypt their keyparts.

   The recommended import procedure separates state import—without relying on KPHs—and commit of the imported state. The KPHs need to be consulted only when the full state has been reassembled.

Remove cloning state erases one or all transient structures from the module.

Query cloning state reports the size or specified parts of the module-resident state. The payload must specify a size query, or a file-part structure to query. If the request specifies a file section, its contents are returned.

The query may be used both on export and import state structures (which are internally not distinguished). Querying during import may be used to test for already present parts. See the wire format for details on specifying file parts and file-size queries.

Requesting an unsupported state structure (i.e., an internal file identifier out of range) returns `CKR_SLOT_ID_INVALID`. If a nonexistent file is queried—valid identifier, but no current content—`CKR_DATA_INVALID` is returned.

If the file-part designation is malformed, missing or otherwise inconsistent in the packed form, `CKR_IBM_TRANSPORT_ERROR` is returned. If the requested range is outside file contents, but the file exists, `CKR_DATA_LEN_RANGE` is returned.

Commands changing control points (CPs) take control point content as a full set of CP bits, and perform one of the possible actions on the targeted set of CPs:

1. overwrite the target CP set with the provided CP payload ("set CP")
2. enable CPs present in the payload in the target CP set ("add CP")
3. disable CPs present in the payload in the target CP set ("remove CP")

Updates fail with CKR_KEY_HANDLE_INVALID if the CP content is invalid (i.e., contains undefined CP bits). Since future updates only add, but not retroactively change the meaning of CPs, using full CP sets from the past will continue to work in future releases. More recent host code—which can always query the number of CPs a given card supports—will be able to set CPs in a binary compatible way, as long as it does not modify CPs added since the cards' release.

Modification of CPs may fail due to CP setup, and *may* is rejected only if the "modification" would actually change the setup. This allows the host to submit the current CP setup without additional checking.

Modification of control points uses N or a single signature, depending on the attribute setup. The default state requires N signatures.

Set domain/card attributes update the host-provided attributes. Only the modified attributes need to be specified. The command returns CKR_TEMPLATE_INCONSISTENT if the provided attribute set is not allowed due to policy—such as specifying a nonexistent attribute, or attempting a change to a value prohibited from modification by another attribute. Note that "inconsistent" refers to the compatibility of newly supplied attributes and the current setup together.

CKR_ATTRIBUTE_READ_ONLY is returned if the specified set of attribute attempts to *modify* a read-only value. We ignore writes to read-only attributes if they supply the current value (i.e., "writing" the attribute would not change it). This special case is supported to allow setting attributes to the then-current value, i.e., allow passing output of an attribute query to a "set" call as-is.

Set clock updates the single, module-global clock as an authenticated administrative action. To monitor time change, two audit events are generated on backends which support auditing, storing the time before and after a clock change.

The command returns CKR_DATA_LEN_RANGE if the time payload is missing or has an invalid size, CKR_DATA_INVALID if the time string describes an invalid time.

Set FCV is accepted only once; it loads the "function control vector", an infrastructure component which restricts functionality (for export control and related restrictions). It is assumed to be issued by infrastructure during the first startup, and therefore does not require authentication. Once committed, there is no command to change the registered FCV; card zeroization removes it.

Without a loaded FCV, the module rejects functional services which require WKs. Services usable without keys, such as random-number generation or hashing, remain accessible. (Since the RNG algorithm is fixed, there is no implied dependency on FCV-controlled cryptographic strength, therefore random-number generation is unconditionally enabled.)

See 10.2.5 for the layout of FCV fields, since we inherit the layout from another specification. The relevant pages are reproduced here for completeness.

CKR_OPERATION_ACTIVE is returned if a different FCV is already present. Passing the same FCV again returns CKR_OK, even if the command itself is ignored during repeated submissions. If the included FCV is rejected due to size or invalid format, CKR_DATA_INVALID is returned.

Zeroize domain uses the domain index out of the command block, therefore requires no targeting (i.e., no payload). It returns a response with empty payload. The affected domain returns to "factory state", losing all key material and administrator certificates. During zeroization, the domain also generates a new instance identifier, and starts again in imprint mode.

The command works on a domain which is not currently active, therefore it only fails if a nonexistent domain is targeted.

Multi-domain zeroize accepts a domain mask, and zeroizes all indicated domains. Non-existent domains are ignored when targeted. The response contains an updated domain mask, containing only domains which were actually zeroized.

Zeroize card resets the card, and all domains, to the "factory state", responding with an empty payload. All key material, administrator certificates, and setup is removed, and a new instance identifier is generated. The currently populated domains are zeroized, and new instance identifiers are generated for them. The card returns to imprint mode.

Note that *card-level zeroization does not erase objects below the EP11 level, such as OA keys, state of the audit subsystem, or other infrastructure secrets.* (In a real HSM, library/application code lacks the authorization to modify or access some of those secrets.)

Zeroize card (system) is equivalent to card zeroization, except preserving the administrator certificate in the first slot of the card administrator list. On systems where an administrator corresponding to infrastructure is registered first—mainframes do this—this administrator may be special, therefore the special-cased query.

Removal of (semi-) retained keys, if (S)RKs are supported, allows an administrator to wipe such a key, without having the full key identifier. SRKs are generally identified by a large handle, which refers to an object fully contained within the backend. Functional use of SRKs, or removal by the controlling user requires the full identifier as proof of ownership. Note that obtaining a list of truncated SRK identifiers is always possible as a functional (non-administrative) query.

This administrative service requires only a truncated SRK identifier, and allows removal of any key referenced by it. It returns `CKR_KEY_HANDLE_INVALID` if the identifier is unknown, and `CKR_KEY_SIZE_RANGE` if the provided "identifier" is clearly invalid.

Even if we allow administrators to remove SRKs, the keys themselves may not be exported.

List card administrator, List domain administrator takes an SKI if querying a specific certificate, or none if requesting a list. It returns the individual certificate corresponding to the input SKI, or a list of SKIs.

Returns `CKR_USER_NOT_LOGGED_IN` if an SKI is requested, but it is not present.

Query device (OA) certificate takes an index into the device (CA) certificate chain, or an empty payload if the number of certificates is requested. The index, if present, is zero-based, with 0 denoting the currently active device key, 1 its parent etc. (On an HSM-backed implementation, this query returns "Outbound Authentication" (OA) certificates from the HSM. Other backends return their own certificates, potentially in a backend-specific format. Test builds may also return OA certificates with a known test root.) Host code must be able to recognize and react to device certificates or chains, the details of which are outside the scope of this document.

The response contains a single certificate if indexed, or a certificate count if no index is provided.

Returns `CKR_KEY_HANDLE_INVALID` if the certificate index is invalid (beyond certificate-chain length). The certificate count query itself does not fail, assuming the rest of request was correct.

Query importer (public) key returns the SPKI of the currently active importer private key. Since this importer key is unambiguous, the query requires no further targeting, other than the target domain.

Query module importer (public) key returns an SPKI, with a concatenated signature, to maintain compatibility with existing migrator applications. Since this signature is redundant, we do not document what it signs—which, for most keytypes, is the full SPKI—and recommend interested parties to verify the response on the signature.

Both queries returns `CKR_KEY_HANDLE_INVALID` if no importer is present.

Query card control points, Query domain control points return the full set of CPs of the requested unit (card/domain). Note that control point sets are different for the two levels of administration.

Control points are packed as a fixed-size raw integer, with interpretation as the bits specified in `XCP_CPbit_t`. The response is otherwise unstructured, not embedded into a structure.

In addition to the enumerated response payload, administrative responses contain the non-payload fields of the originating administrative block, including targeting.

Query card attributes, Query domain attributes return the packed array of integer attributes supported in the targeted unit. All supported integers attributes are returned.

When querying card attributes, an optional parameter may be provided, which describes domains on which *compliance attributes* are to be aggregated as a card-level compliance setting (see format in the wire section). If no parameter is provided, the reported compliance attribute is the intersection of domain-compliance attributes on all domains with loaded WKs. (Since our security policy specifies card-level compliance this way, this behavior is the default.) If the domain mask is invalid, or the field is malformed, `CKR_DATA_INVALID` is returned, with a corresponding reason code.

Query current WK, Query pending WK return the WK verification pattern of the active and imported (next) keys in the targeted domain, respectively. Both fail with `CKR_KEY_HANDLE_INVALID` if the properly targeted domain lacks the queried key.

Query WK origins returns the WK verification pattern of an imported WK, and its key parts, if they were used. The returned payload contains key VPs of the full key, followed by those of key parts, the latter in arbitrary order. If the key has been created/imported without key parts—such as a pending key filled by "create random WK" or imported through development extensions—only the full key VP is returned.

The query fails with `CKR_KEY_HANDLE_INVALID` if the domain lacks a pending (next) key.

Query FCV returns public portions of the controlling FCV, if active (format TBD). If the backend has no active FCV, the return code is `CKR_OPERATION_NOT_INITIALIZED`.

Query audit state returns the number of audit event records, or one specific record, depending on its input parameters. When no payload is provided, or the index 0 is requested, the response only contains the number of currently held records (see section 9.4.5).

When an index or a targeted audit-event state is passed with the query, the response is a single audit event, or an error return code indicating the failed lookup.

Administrative return codes In addition to command-specific return codes, administrator commands may return the following values, in decreasing priority:

1. `CKR_PIN_EXPIRED` if the active transaction counter is already higher than the one included in the command.

   Queries ignore the transaction counter, and may not return this return code.
2. `CKR_SLOT_ID_INVALID` if a domain-specific query/command targets a nonexistent domain.
3. `CKR_FUNCTION_CANCELED` if settings—control points or attributes—prohibit execution of the service.
4. `CKR_FUNCTION_NOT_SUPPORTED` if the requested administrative service is invalid, or not available.
5. `CKR_SESSION_CLOSED` if the target has already left imprint mode, and the request targets an imprint-only variant—such as logging in without signatures
6. `CKR_SESSION_HANDLE_INVALID` if the target is in imprint mode, and the request may only be serviced outside imprint mode
7. `CKR_USER_NOT_LOGGED_IN` if not all signer/s signing the command are allowed to administer the target—card or domain.
8. `CKR_TEMPLATE_INCOMPLETE` if the command has less signers than required. Note that too many signers are tolerated, and not reported as an error.
9. `CKR_KEY_CHANGED` if signatures are present, when not needed, or the signature field is empty, when it should be present.

   Some of the services tolerate extraneous signatures under specific conditions, even if otherwise not needed, for historical reasons. Obviously, these additional signatures—or lack thereof—are not security-relevant (as the base service in its current form is allowed without signatures).
10. `CKR_DATA_LEN_RANGE` if required payload is missing, or payload is provided for a command without one.
11. `CKR_DATA_INVALID` if payload (formatting) is inconsistent with the requested command, if payload is present.
12. `CKR_SIGNATURE_LEN_INVALID` if at least one of the signatures is malformed, including other signature-packaging errors (not just raw signature size).
13. `CKR_SIGNATURE_INVALID` if any of the signatures is invalid. All signatures are verified, and verification fails even if there would be sufficient valid signatures to complete the command (i.e., only supply valid signatures).
14. `CKR_IBM_INTERNAL_ERROR` if crypto primitives fail. The host should not encounter this error, and logs should be checked for further diagnostics.
15. `CKR_DEVICE_MEMORY` if transient memory is not available (which should not happen).
16. `CKR_IBM_READONLY` if the backend is not allowed to update persistent databases, and state-changing commands are not allowed.

    Currently, only *concurrent driver update (CDU)*—backend firmware updates without service interruption—may cause a module to enter readonly mode. Alternatively, a test function exists to force the same in diagnostics builds.

17. `CKR_SLOT_ID_INVALID` if targeting is inconsistent: if domain-targeting requests feature inconsistent domains in their—redundant—domain fields, or if a card-level command features a non-zero domain in its request.

    While the same PKCS#11 error code is used to indicate domain mismatches, reason codes differentiate between the exact logical conditions.

Command-specific failures are lower priority than the above generic ones. Note that some of the return codes "override" ones EP11 does not return otherwise. Others may be returned under other conditions for functional calls, but they are unambiguous when returned by administrative traffic.

List of returned WK verification patterns is returned by certain WK-related administrative services (import, commit, query WK origin). These lists contain the verification pattern of the full WK, and that of its constituent keyparts. Keypart verification patterns are in arbitrary order. The list is returned to allow each KPH to verify that their expected verification pattern is present in the key, allowing procedural control over key activation.

If the WK has not been assembled from multiple parts—was generated internally, set directly through a test function, or imported as a single "keypart"—only the verification pattern of the full key is returned. We do not currently distinguish results based on import source; differentiating an internally generated single-part key from an imported one would be a trivial extension—such as an additional flag bit—if required.

8 Key cloning

8.1 Cloning mechanics

Keystores are cloned through copying a single transport wrapping key between domains. Cloning is publickey-based, using structures that accommodate single or multi-part messages (PKCS#7 envelopes). When two domains' transport wrapping keys are synchronized, they are identical for the purposes of EP11 commands, since EP11 objects are tied to the transport wrapping key that generated them. EP11 commands therefore may be dispatched to any of the synchronized domains or cards.

Cloning relies on atomic messages, since *PKCS#7 structures retain all necessary intermediate state*, and therefore may be constructed or parsed in one pass, or assembled incrementally, but used as a single atomic messages. Administrator signatures are applied to unambiguous command blocks. We derive our message formats from PKCS#7 structures, with implicitly known signer certificates and no CRLs [PKC93, Hou04]. More specifically, we use derivatives of the following PKCS#7 formats:

1. `SignedData`, containing clear payload, with an arbitrary number of signers. We apply signatures as plain SignedData structures, without any other PKCS#7 encapsulation.

   We can implement our scheme without CRLs, as certificates would be readily available. Therefore, raw SignerInfo structures—identifying a signer (key), signature algorithm and signature—are unambiguously verifiable.

2. `EnvelopedData`, containing RSA or EC-encrypted data, for an arbitrary number of recipients.

   Each recipient contributes a certificate, and the resulting `EnvelopedData` contains both per-recipient (encrypted) and common (clear) content. Each recipient can decrypt a specific part of the encrypted payload (i.e., one `RecipientInfo`).

3. `EnvelopedData` *with ECDH RecipientInfo* if EC transport is supported. This format is effectively the EC equivalent of encrypted data, although its symmetric encryption is indirect, includes key derivation and symmetric encryption—unlike RSA [SEC00, 5.1].

   Note that in our use, we perform one-pass ECDH "key agreement", which may procedurally replace RSA encryption, as their use pattern is comparable. In this mode, the recipient uses a static public key, and therefore the transfer—other than transporting the recipient public key—is single-directional, similar to RSA-based key transport [BJS07, 6.2.1].

   Note that *the explicit authentication inherited from our administrative traffic provides security assurance about communicating with the proper recipient.* See [BJS07, 6.2.1.5] for security considerations.

We may combine the two formats. EnvelopedData derivatives are used when moving keyparts, targeting either importer or KPH private keys with sensitive data. We append administrator signatures as a collection of raw `signerInfo` structures (but not use proper PKCS#7 encapsulation) into the "signatures" field of the signed packet. Encrypted data, packaged as individual RecipientInfo's, are collected into the administrative "payload" field.

Note that the required PKCS#7 subtypes all allow multi-valued instances, i.e., multiple signers or recipients.

The cloning process has minimal requirements on host (or administrator) infrastructure: administrators need to support RSA/EC signing and verification; KPH need to be able to en/decrypt data (keys). Encryption and signing flows are identical for RSA and EC keys, as we rely on 1-pass ECDH, which uses static public keys for one of the parties (and therefore, resembles RSA instead of symmetric ECDH).

Cloning may involve multiple administrators or multiple secret shares, in any combination. *Multiple administrators* are needed when operational security requires dual control over administrative action, such as adding new cards to the system, but not necessarily key archival. *Secret sharing* is needed when keys themselves are under dual control, and no single individual has access to a key outside, either in clear or encrypted form.

Note that logically we may combine multi-signer and multi-recipient messages, but implement them with parallel signatures or multiple, single-recipient messages. *We allow such parallelization, and accommodate both single-message and multi-message commands, as long as they may be unambiguously combined.* In the following examples, we assume that single-message, multi-component structures are used. If administrators are distributed, for example, one may embed multiple `SignedData`'s containing the same payload, and combine them during command processing (which does not change security, and it is unambiguous).

The only long-term persistent, card-resident cloning-related data is a list of administrator identities, i.e., X.509 certificates. Since the modules have no way of establishing trust to external CAs, certificates are used only as a portable way of representing public keys, and are not otherwise checked for validity. *In order to be able to audit modules, administrator public keys must be available for host queries.* Having an public set of administrator keys allows one to audit modules and their administrators directly.

Cloning happens beneath the level of PKCS#11, and it is not represented at the PKCS#11 level. Transport wrapping keys are not PKCS#11 objects, and are not represented in PKCS#11 calls.

The cloning process uses a disposable importer key for a single transfer, and therefore must be performed pairwise in groups of multiple cards. *We include a verification pattern with the transported key*, which serves as an integrity against trivial errors. The integrity check is not authenticated, and must be a one-way derivative of the transported key (i.e., it is sufficient to use a cryptographic hash function). When transporting multiple keyparts, each keypart will have an additional verification pattern . (See expanded section on verification below.)

In the following descriptions, by "all administrators" we refer to the necessary number of administrators (so that the module accepts administrator signatures). The actual number may be different from the number of registered administrators (it will be if redundant administrators are stored for resiliency).

We rely on administrator transactions providing their own transaction counter for replay-protection, and do not include that in our description. In a few cases, we actually introduce our own replay protection as a side effect of our cloning scheme.

8.2 Cloning without secret sharing

In the *simple cloning scenario, without secret sharing*, a module may export its transport wrapping key for any single RSA public key, as long as the administrator (all administrators) sign the receiving public key. The public key will be the public part of an importer key, generated by the cloning target module as a single-use importer.

Administrators will need to build a PKCS#7 `SignedData` with the public key as payload, and import it to the source module. Since PKCS#7 `SignedData` signatures apply to the same payload, and are verified in parallel, relative order of signatures is not relevant. Since PKCS#7 SignedData contains a clear payload, each administrator can inspect the SignedData structure passed to him, therefore administrators may (and should) cross-check each other.

Actual cloning steps of single-part key cloning are the following:

1. Target module generates importer keypair. It exports its public key to administrators.
2. Each administrator signs the public key, authenticating it for the exporting module. Each administrator appends a new signature—i.e., `SignerInfo` structure—to the same payload.

   Similar to what's shown in Fig. 9, admins must build the `SignedData` sequentially. There are no ordering requirements, since PKCS#7 signatures are verified in parallel. The only requirements are that administrators can pass around binary blobs reliably (in the end, the modules cross-check everything).
3. Import the signed importer public key to the exporting module. Administrator public keys must have been deposited to the exporting module, and therefore the entire PKCS#7 `SignedData` may be verified atomically.

   If there are problems with the PKCS#7 structure (mismatched administrator identities, invalid signatures etc.), the export request is rejected.
4. If the export request has all necessary signatures, the source module encrypts its transport wrapping key under the public key embedded in the request, and returns it to the host.

   The transport wrapping key, when exported, may be inspected by all administrators. It may be decrypted only by the target module, since the only instance of the importer private key resides there.
5. Administrators repeat the signing procedure (Fig. 9), this time signing the exported encrypted transport wrapping key. Once all signatures have been appended, the `SignedData` may be passed to the target module.

   The order of administrator signatures is unimportant, as mentioned before.
6. The target module verifies the `SignedData` as an atomic operation. It must have administrator identities (certificates) at this point.

   The wrapped transport wrapping key is recovered by a custom mechanism of `UnwrapKey`, using regular interfaces. Obviously, this special `UnwrapKey` does not return data.

   If all required administrator signatures verify, and the payload decrypts with the active importer key, one can replace the target's transport wrapping key with the decrypted key. At this point, the two modules' transport wrapping keys have been synchronized. (Session synchronization must happen in addition to transport wrapping key cloning.)

   After an attempt to import a key, the importer private key is destroyed. This prevents replay attacks of any kind, rendering externally remaining cloning tokens useless.

The cloning process is essentially identical if a single administrator is allowed to make changes. In this case, the `SignedData` contains only a single signature, all other details above are identical.

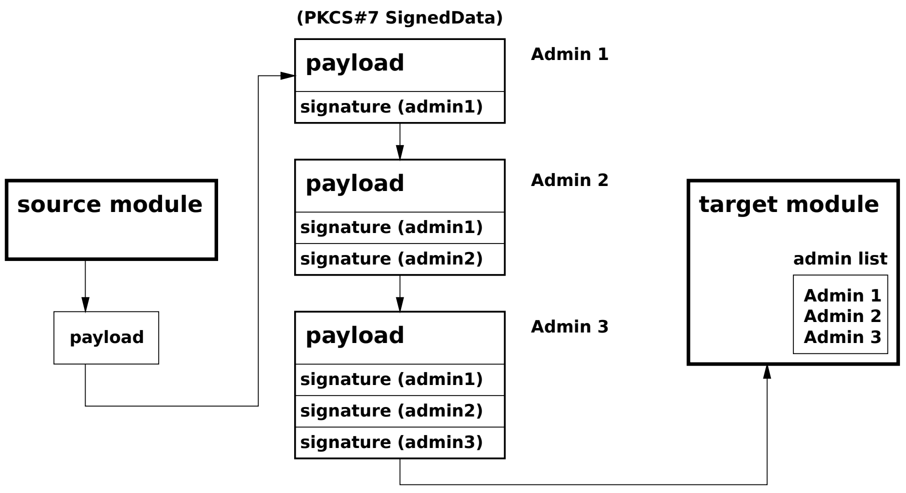

Figure 9: Cloning: signing during export data flow, multiple administrators

8.3 Cloning through key parts

When keys must not be revealed to any single outside—i.e., non-HSM—party in clear or encrypted form, the payload in the cloning process changes, while the administrator signing procedure is unchanged. In this setup, we require key part holders (KPHs) to encrypt key parts with their own RSA keys. (Without encryption, hostile host administrators could reassemble the keys, capturing keyparts directly from the device bus.)

Relying on one-pass ("asymmetric") ECDH key agreement, one can implement the same "encryption" scheme with EC primitives if the receiver uses a static keypair, and the recipient an ephemeral one. This "static-ephemeral" ECDH is procedurally interchangeable with RSA, and offers assurance comparable to RSA key transport [BJS07, 6.2.2.3].

When exporting keys parts, the source module needs to receive a list of KPH certificates, which must be signed by administrators. This list can be encapsulated inside a PKCS#7 `SignedData` structure. (Perhaps counter-intuitively, KPH certificates would need to be transported as the signed data, not inside the PKCS#7 certificate field.) As described previously, the `SignedData` structure may be built externally, and may be processed by the exporting module in an atomic fashion.

Note that the source module does not need to retain KPH certificates, as the entire certificate list is submitted for each cloning request.

Once the source module has verified administrator signatures, it can extract KPH certificates from the PKCS#7 payload. The transport wrapping key is then split into the necessary number of key parts, and each part is encrypted by the corresponding KPH's public key. The resulting *encrypted key parts* are encapsulated in another PKCS#7 structure, an `EnvelopedData` compound. This structure may be exported from the module as each part is useless for anyone except at most one of the KPHs.

When the `EnvelopedData` is revealed to the host, each KPH may decrypt his part without learning anything about other parts. KPHs then may retain their key part in the clear or inside trusted storage (such as TKE). These key parts are unused until the next key part import.

Importing key parts is a two-step process. The target module must generate an importer keypair, and present it to the outside administrators (and keypart holders). First, KPHs encrypt their key parts for this importer key, and reassemble those into another `EnvelopedData`, this time encrypted for the importer key. Administrators then sign the resulting `EnvelopedData`, encapsulating it within a `SignedData`, which then may be passed to the target module.

Once the `SignedData` structure containing key parts is returned to the target module, it first verifies that admins' signatures are correct, and that the payload is PKCS#7 `EnvelopedData` (targeted to the importer key). If all these conditions are met, the key parts decrypted from the embedded `EnvelopedData` may be combined to form the new transport wrapping key.

Similar to exporting key parts, the target module need not retain KPH certificates. All necessary certificates must be part of the cloning request structure (inside PKCS#7 structures), and only administrator certificates are not disposed after processing the request.

During regular key migration, when one backend provides keys to others, one may embed auxiliary information within individual KPs. Such auxiliary information makes the scheme immune to hostile KPHs, as they would be included when honest KPHs feed back their KPs to the target module. (A hostile KPH obviously could not compromise KPs from honest ones.) If keys are individually generated by untrusting, independent parties, such as in the current TKE scheme, we may not rely on these

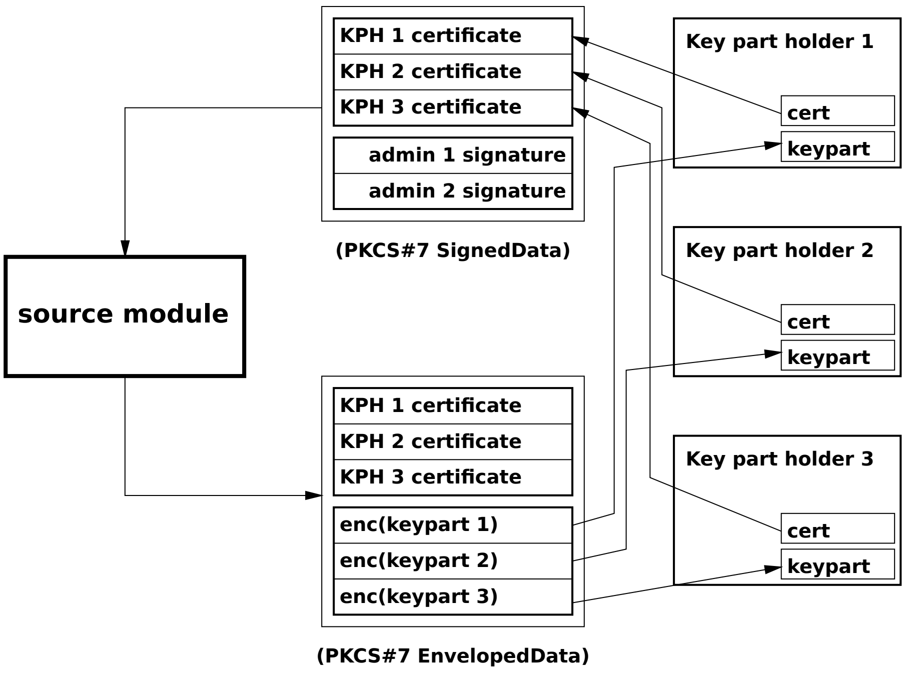

Figure 10: Cloning: logical flow of key export+import

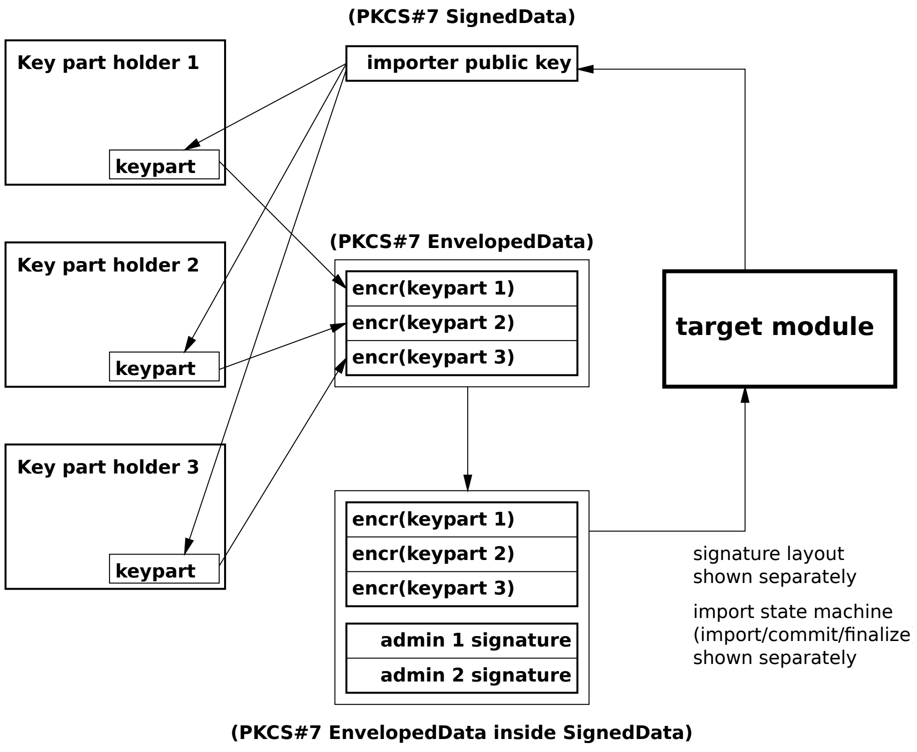

Figure 11: Cloning: logical flow of key part import

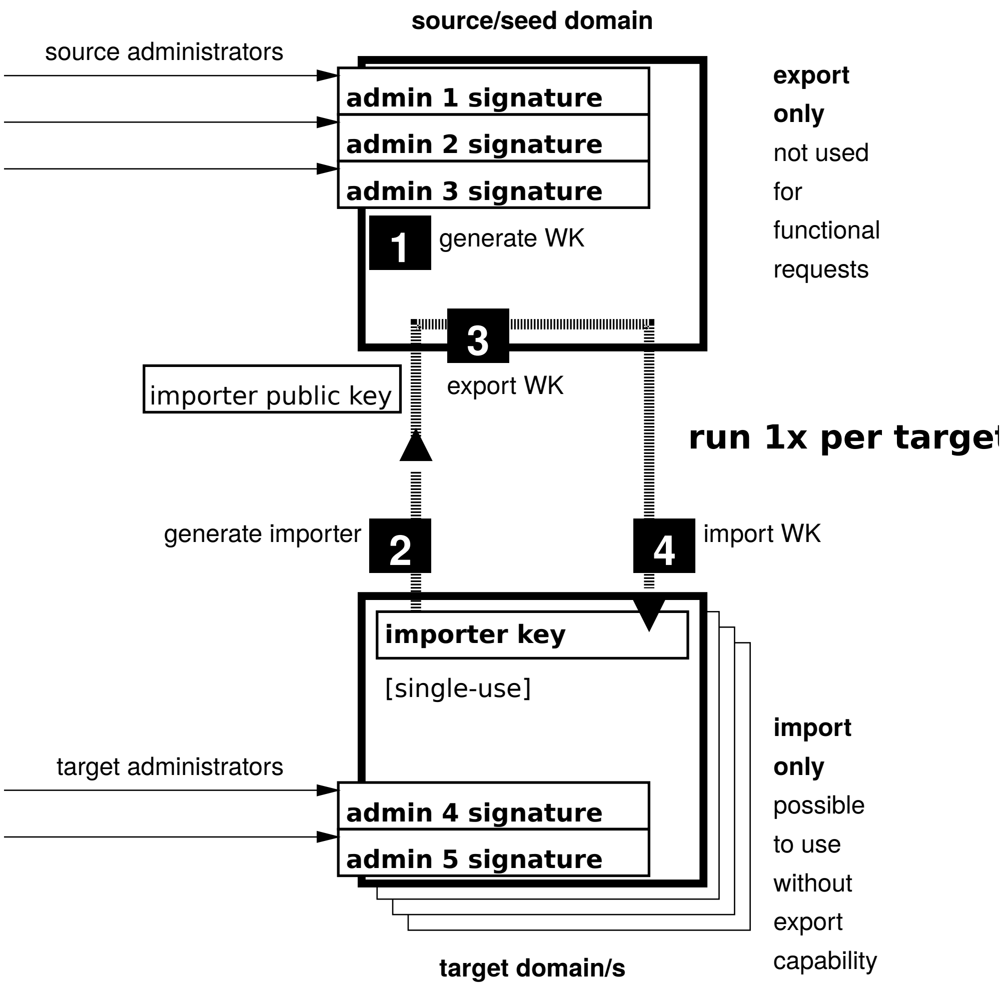

Figure 12: Cloning: key transport without intermediate key decryption

additions, and we need to tolerate these weaknesses during the legacy-transition period.

During the cloning procedure, sensitive information is never revealed outside modules. Exporting and importing modules authenticate requests, exporting and accepting only trustworthy public keys. The exported, encrypted secret is not usable outside the importer, specifically without the importer's single-purpose, single-use importing key.

One should note that the both source and target modules may unambiguously differentiate between combinations of single/multiple administrators/keypart-holders. The current implementation is therefore a a pair of non-PKCS#11 cloning calls, one for export and one for import. Cloning calls are polymorphic, selecting mode based on data presented to them.

8.3.1 Cloning directly between domains

If procedural restrictions allow it, WKs may be transported between two domains without intervening decryption by combining a direct Export WK and one Import WK for each targeted domain (Fig. 12). *This mode may only use a single asymmetric-encrypted key, no keyparts, and therefore requires both source and target domains to allow single-part key transport.* Since each invocation encrypts for a non-extractable private key resident in the target HSM, which may not be replicated, *splitting a key would not increase security* as the single message may be decrypted in only one location.

8.3.2 Data transported during cloning

Serialized state is exported from the source module in two related data files, and imported into the target in two files, one of them used verbatim. All files are encoded in a straightforward, self-describing tagged multisection format (see the wire section for tag listing and file composition).

The following files are used during export and subsequent import:

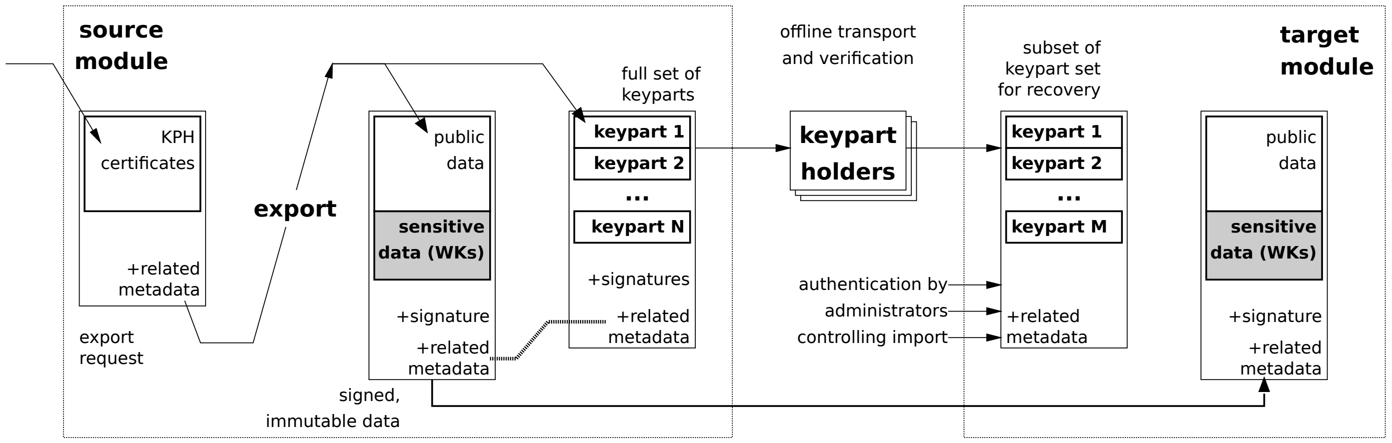

Figure 13: Cloning: filesets used during key export+import

1. Request to export, a similarly tagged multisection file describing the intended export mode, potentially including the intended KPH recipients.
2. Serialized module state, excluding transportkey-parts. This state contains non-sensitive data structures in cleartext, and sensitive data—any present WKs—as a single, encrypted region. This fileset must be passed to the receiving module without modification.

   The serialized state is signed by the originating module. Note that its import itself is signed by administrators, therefore verification of the signature on the state—during import—is actually redundant.

   Since decryption of the encrypted portions requires collaboration of the necessary minimum of KPHs, *the serialized state file is not sensitive.*
3. A full set of parts of the transport key is exported in an auxiliary data file. This set of keyparts contains encrypted sections, all decryptable only by the targeted KPH. The entire file, as well as—for historical reasons—all keyparts are signed by the exporting module.

   Metadata included in the keyparts' file allows unambiguous identification of the corresponding serialized state: identifying fields such as creation time and an export-unique salt are included in both files.

   Exported keypart files may be discarded upon loading into the recipient KPHs. They are no longer used by EP11 backends.
4. A subset of keyparts, all encrypted for the current importer key within the target module, each keypart encrypted by an authorized KPH.

   If the set of KPs is sufficient, the receiving module may decrypt and reassemble the transport key, and then decrypt sensitive data within serialized state, then synchronize all internal state of the target module to that of the source one.

As show in Fig. 13, export generates two files. Portions of the keypart set are archived by KPHs, and that set need not be retained after archival. During import, the subset of KPs used for recovery is similarly constructed, and must accompany the original serialized state.

Exporting and importing files is supported in a simple file-transport method, which allows incremental reads and writes, allowing cloning even if transport limits per-request data sizes.

Both exported state and generated keyparts are signed by the originating module, including the full certificate chain of the originator (see wire section for details).

8.3.3 Simultaneous import into multiple domains

Exported data from a single domain may be imported back to a different domain or multiple domains, if the import request designates it as such. The corresponding state to import must contain data from a single domain—this may be unambiguously determined from section headers. The section designating multi-domain import must contain the domain mask of the intended target domain/s. *All previous data of the targeted domain/s will be replaced upon successful import,* even fields missing from the newly imported state. See wire rules about the `MULTIIMPORT_MASK` section type for encoding.

When module-level sections are present in state submitted for multi-import, they are ignored. Note that restrictions on the data exported allow the original source module to skip module-level sections from exported state; see the `STATE_SCOPE` section for restricting export-file contents.

Since importing into multiple domains is technically a domain-level operation, this import type *could* be extended to allow importing into domains with an identical set of *domain* administrators. This possibility, however, is not currently supported: *multi-domain import must be authorized by module-level administrators* (even if importing into a single domain).

When attempting import into non-existent domains, requests to import are rejected. While in some other administrative cases we tolerate—and ignore—requests for future constructs not or not yet supported by the backend, we require very specific requests for intrusive operations which overwrite domains.

8.4 Key verification patterns

When exporting keys, we include two integrity fields within each encrypted KP, which KPHs can recover and use to verify integrity. Verification patterns are hashes of key material (with some predefined formatting), and are assumed not to reveal information about the key. We include the verification pattern of the entire—reassembled—key, and one for the individual KP.

When a KPH recovers a KP, they will need to store recovered verification patterns along with the KP. Typically, this could happen within the KPH's own secure signing/encryption device, such as a smartcard or an HSM.

When importing key material, KPHs may use their KP's verification pattern to protect against procedural errors. The encrypted KP still needs to include the entire-key verification pattern, which the target module verifies.

When a target module reassembles KPs, it recovers and verifies overall integrity, which all KPHs will include (even if they can't use it themselves). If the verification pattern of the reassembled key does not match *all* of the KPHs' attached signatures, the backend rejects the imported keyparts.

Note that existing legacy MK-parts setup scheme, which combines independently generated, unstructured random values, is not suitable for aggregate integrity checking, as keyparts are not available simultaneously before key import is completed. If such a key is recovered from KPs, all KPs would contain only raw key material, but no aggregate verification patterns. If all KPs decrypt to such VP-less material, the backend assume that the KPs are all externally generated, and accepts this compound without an aggregate verification pattern. We assume that at least one honest KPH would have included an overall verification pattern if it has been supplied during export, even if rogue KPHs would not. Similar schemes are used in distributed storage security schemes, where any honest party alone may expose malicious—but authenticated—actions of hostile entities [WOW08, 5].

As an integrity check after importing KPs, the backend returns the verification pattern of the imported key, and those of its constituent KPs. Administrators must verify that KPHs' approval is granted before the reassembled key is activated, when they check the list of verification patterns for their own. If a KPH does not notice the verification pattern in the latter list, it MUST terminate the import procedure witholding the signature required for activation.

Note that our use of key verification patterns differs from that of PKCS#11 key-checksums. We make the distinction unambiguous, wherever applicable. Since WK and keypart management is outside PKCS#11 scope, the two types are never ambiguous.

8.5 Cloning fundamental assumptions

When describing cloning, we make the following assumptions about control and data flow:

1. Our scheme is designed to protect against an arbitrary number of dishonest administrators, as long as the system threshold disqualifies them from submitting commands signed *only* by dishonest administrators.

   In practical setups, one would most likely administer with a majority vote of three (3) administrators, and one could therefore tolerate a single rogue administrator.

2. We need to reject cloned keyparts as long as a single honest KPH is active in the system. We assume that "mainly honest" administrator signatures will be applied only to KPs from the the (see above)

3. Within system limits, the performance of administration is stable, does not depend on the number of registered administrators. Therefore, *registering mainly-inactive administrators for disaster recovery purposes is possible*.

   In other words, one may register administrator certificates reserved for infrequently accessed, archived administrator-signing keys. Not used during regular operations, such keys could allow recovery if the frequently active keys are lost.

4. We provide source-to-KPH and KPH-to-target integrity and confidentiality, through a combination of signatures and public-key encryption. *Our scheme does not provide end-to-end guarantees between source and target modules.*

   Note that in certain practical setups, administrators would only partially overlap for source and target modules. In such an environment, one would not be able to provide generic end-to-end authentication between the two modules.

5. *Without N-of-M key reassembly, our scheme fails if KPHs' key storage is not highly available,* as loss of any single KP will render the exported key useless. *We require KPHs' environments to provide reliability while they possess their KPs.*

8.5.1 Restricting cloning

Depending on requirements of the host environment, certain capabilities must be switchable. Many of these settings are further meta-controlled by a switchability flag, removing even the controlling administrators capability to change the state. Once removed, the capability may not be enabled without restarting/reinitializing the module (selection depending on the meta-control-bit restricting it).

Enable key export is a capability one may wish to remove from "leaf" modules (such as those deployed to remote offices). In such a setup, only a central group of modules would be allowed to push a transport wrapping key to leafs, but those could not export them any more.

Enable key import allows cloning to accept a new key. If this item is disabled, the module will reject further attempts to change its transport wrapping key. (Obviously, one could always reinitialize the module.)

Allow single-part keys could be left enabled from CA-type applications where key dual control is not feasible or not desired. Lower-level CAs or server keys that are easily replaced could be cloned in single-part messages, since the procedure itself is more straightforward.

Systems where module cloning must remain under dual control would disable this capability, allowing key transport only for at least $N = 2$ key part holders.

Note that this capability combines individually with import and export. Once activated, it applies to both.

Module initialization is a convenient point to set up cloning restrictions. One would expect that the host driver would set an administrator-controlled profile to each module before activating it.

As one example of a financial (PIN-processing) application, one would set up a profile similar to the following:

1. *Require key import*, i.e., don't activate the module with a random transport wrapping key
2. *Disallow single-part keys*, forcing dual control on cloning
3. For central Master-Key backup nodes, *allow key export* and possibly *disallow key import.*

   Conversely, for non-central nodes, *inhibit key export* and *allow key import.*

As another example, a timestamp server or a lower-level CA that can replace its keys easily, could operate in the following setup:

1. *Don't require key import*, activating the module immediately with a random transport wrapping key
2. *Allow single-part keys*, if the CA operates without dual control.

Key import and export may or may not be relevant in this setting.

A semi-centralized high-assurance setup issuing digital signatures may be split into two parties, one "key server" generating and auditing private keys, and multiple "leaf signer modules" using them. In such a setup, one could reliably audit flow key lifecycles, and deploy leaf signers without them acting as key sources.

1. An initial, possibly offline activity shares WKs between all servers.
2. *Key servers generate and export private keys,* generating key objects and exporting the local audit history to document origins and lifecycle of the generated keys.
3. *Leaf signers may import, but not export private keys,* relying on domain attributes, not only on key usage restrictions.
4. All participants could prohibit explicit use of secret keys, relying only on WKs, and preventing use of the signing infrastructure for unrelated activities.
5. Setup of key servers and leaf signers is switched to non-modifiable, preventing further changes.

To establish trust in modules, an unrestricted host service should enable admins and key part holders to query current card configuration. *Since most critical attributes are controlled by attributes preventing their change, applications may verify that the current setup may not be changed without complete zeroization.*

9 Security assumptions and rationale

9.1 Sensitive state storage

9.1.1 Authenticated encryption format

Sensitive state is encrypted and authenticated using a combination of AES/256 and HMAC-based signatures, using a MAC key derived from the controlling WK. Blobs are wrapped in an *Encrypt-then-MAC* structure [BN08, 1.2, 4.3] [Kra01, 4.1]. Note that unlike protocols with flexible orders of encryption and authentication, we only support encrypt-then-MAC, and do not support falling back to other ordering [Gut14, 4, Security Considerations].

Since blob verification/decryption is performed in a single pass, it is by construction immune to attacks based on fragment reassembly [DP10, 3.1.3] or compromising the reuse of cryptographic state [RRDO10, 1]. Since we do not react to data with an invalid MAC, blob decryption does not provide decryption oracles [SF13, 2.1]. As in an encrypt-then-MAC structure, where the MAC is applied to encrypted contents and not embedded within ciphertext, MAC verification—using entirely host-visible (amounts of) data—does not provide timings usable for side-channel attacks (unlike TLS, cf. [AP15, 2.3]).

If the backend resides in a different address space, and host code may not modify the request during parsing, *blob processing is immune to time-of-check time-of-use attacks—i.e., "double fetch"—by construction.* In these cases, since all reads of the request access backend-local memory, the backend will work with a consistent request structure even if the host is hostile, as the latter may not modify requests between backend accesses [JC13, 3.2].

Depending on configuration, our backend *may* execute in the address space of the host component, but this is not the case in high-assurance environments—such as HSMs—or "soft-HSMs" where the backend resides within a dedicated partition. *Since production EP11 environments execute in different address spaces from their callers, we acknowledge and ignore the problem in caller-accessible backends* without attempting to fix it. If protection is to be added later, we note that only memory accesses need to be identified in our stateless model, which is considerably easier than detecting double checks at a filesystem interface [PG12, 4].

Since sensitive state is signed last, after encryption, our verification-then-decryption process is assumed to be immune to practical attacks on decryption/padding/decryption oracles [FP13]. As a side note, since encrypted content starts with length-describing fields, unpadding of the last encrypted block is superfluous, and not required for proper operation.

9.1.2 Authenticated, non-sensitive state

Non-sensitive state is currently restricted to authenticated public keys—SPKIs—which are published, but are bound to attributes. These objects share MAC keys with their private counterparts, derived from their controlling WKs.

9.2 Security rationale

9.2.1 Attack scenarios

Physical protection Depending on its deployment, the backend *may* inherit physical-security features, primarily active tamper protection, from the environment it is deployed in. We assume automated tamper responses, independent of software intervention, therefore *we implicitly assume tamper responses would be processed without active action of our backend*. With this disclaimer, we assume to inherit physical protection when possible, but it is orthogonal to our own protection features (such as defenses against faulty hardware).

Note that all IBM HSMs feature hardware-based tamper-protection, circuitry which is entirely independent of the firmware it is loaded with. Our assumption is based on such firmware-independent tamper protection, and may be changed if our backend is deployed in less capable security modules.

The only tamper-related action performed by our backend, when deployed on IBM HSMs, is reaction to card removal. While removing a module from the machine hosting it is not considered a tamper event, it is logged persistently—"external warning" indicator—and may be reacted to when tamper-aware firmware is subsequently booted. *We maintain an administrative setting to control such reactions, and manage it when running within a suitable HSM*—we wipe persistent state after card removal if so set (see the `XCP_ADMM_EXTWNG` for details).

Replay attacks Due to the mainly stateless nature of Enterprise PKCS#11, *replay attacks* are generally ignored as a threat: a stateless CSP is in a state of permanent replay, and replay-related problems MUST be separately managed if necessary [FG17, 1.2]. Note that the citation also shows why dispatching requests across multiple, related backends would prohibit some naïve replay-countermeasures, and replay protection is better made more explicit at the protocol level.

There is a limited number of scenarios where requests—generally, state-changing commands—are specifically prohibited from replay:

1. State-changing administrative commands must include an increasing, administrator-specified transaction counter.
2. *Importing state or WKs* uses a single-use, module-internal private key as an importer. These private keys are discarded upon first use, or when a replacement key is generated, which prevents replay attacks against import-related functionality.

As a special case of replay protection, *instance identifiers* are created randomly, used as "salts" for cases where the module-internal setup *might* repeat after zeroization. Including instance identifiers prevents past requests from being accepted after zeroization, if the environment is restored to an otherwise identical state. As an example, if a domain is zeroized, then repopulated with identical administrators as before zeroization, past administrative traffic could be replayed—since module identification, domain number, or administrator keys are unchanged. However, since the domain number is "salted" through adding an instance, valid administrative traffic for the same setup is still prevented from replay, as commands must include domain instances.

Note that size of instance identifiers is limited—several bytes—therefore there is a non-negligible chance for collision, if the operation is performed sufficiently frequently. Since zeroization is disruptive, and would be procedurally monitored—or even persistently logged—in a practical setting, we ignore the possibility of collisions of instance identifiers. Note that even in such cases, the administrators capable of triggering a sufficient number of zeroizations would be then become capable of launching a replay attack against themselves. We acknowledge this vulnerability without attempting to fix it.

Signing oracles To prevent signing arbitrary data, Enterprise PKCS#11 backends insert nondeterministic data to signed responses, to prevent signing data controlled entirely by the host, or more generally, to prevent signing data considered to be known the host.

At the modest cost of random-number generation overhead, salting responses also increases the complexity of any attacks related to hash collisions, such as when audit logs are authenticated as as hash chain, or when hashing is implicit in digital signatures [Fil13, 3.1.2].

- Sensitive objects include an internally generated, random IV, both during key generation or key import. The IV is included in the MAC calculated over the entire object, and therefore doubles as salt of the authenticated object.
- Authenticated, non-sensitive state—MACed public keys—add salt to prevent signing entirely user-controlled data. Note that internally generated public keys are basically not user-influenced—except for the optionally specified public exponent—but public keys imported to be MACed are.
- Audit records include a minimum amount of salt, preventing any host entity from advancing the audit state only based on known or host-controlled data. The amount of salt varies by usage—see the wire section for details—with only a specific minimum bitcount enforced.

  Note that salt is explicitly included, in addition to any host-visible, predictable content which is ignored when considering randomness—such as timestamps or sequence numbers.

  Note that audit records are based on hash chains, and are not themselves signed. Salting prevents user-controlled advances of the audit chain itself; it is listed here as it resembles defenses against signing arbitrary host-controlled content.
- Audit records, when signed within an administrative response, include additional salt within the response block. This prevents issuing an OA signature on fully known input: the then-current audit state must be considered to be known.

In addition to the above salting/randomization steps, *attribute-bound key wrapping includes a module-generated, random IV,* and possibly a transient transport key, both generated by the originating module. Obviously, direct access to the signature key may be procedurally possible, therefore randomization of AB-wrapped enclosures is only relevant if the host otherwise restricts access to the AB key. Against attackers who have indirect access to AB-wrapping functionality, randomization described above prevents signing fully user-controlled data.

The minimal amount of salt outside host control—beyond 64 bits—is expected to realistically prevent attacks on event randomization, matching real world experience of runtime randomization [CCF$^+$16, 5, "Randomization entropy"].

Timing side channels The backend intentionally avoids data-dependent dataflows, providing constant-time comparison for cases where timing may leak signature or plaintext information [Por16]. As the most obvious example, MAC verification of host-resident, authenticated state is always through data-independent operations. (Note that our constant-time comparison also maintains a uniform memory-access pattern to minimize any secondary leakage, such as through cache behaviour.)

Note that the underlying crypto provider also provides constant-time implementations for many algorithms, such as PKCS#1.5 unpadding [BFK+12]. These timing countermeasures are not further discussed here.

Maliciously malformed requests Our attack model, mirroring HSM trust assumptions, expects arbitrary combinations of accidentally or maliciously misbehaving host code. The *backend distrusts anything appearing in requests*—i.e., on our wire interface. Note that our *distrust extends to host library code, even OS/infrastructure libraries,* and therefore our security assumptions conservatively account for attackers with essentially unrestricted level of access to—the wire interface of—the hosting HSM. Obviously, most server environments include serious host-based protection, which we completely ignore. This prudent level of distrust contrasts with commercial products assuming mainly cooperating environments [KCR+10, 3.3], and is prudent in embedded systems interacting with untrusted request sources [BGJ+12, 6].

We do not specifically distinguish between malicious or simply buggy host code, and only mention malevolent modification in the following paragraphs. In practice, our test processes combine both completely random requests [ADG+92] and maliciously malformed ones [MCM06, 4.2], with the latter incorporating specialized knowledge of some of our data structures. Our regression-test tools also apply targeted mutation to increase coverage by incrementally modifying known-good starting requests [BGM13, 2.2].

At the first level of protection, we assume attacks by host code who may not forge signatures on backend-generated data—i.e., blobs or signed SPKIs. Typical server-centric applications, lacking access to WK management, would be capable of only this level of malice. Backend code assumes malice at all levels, therefore any formatting errors of the first level are reported and rejected as "expected" errors—i.e., produce no additional logging, other than reporting errors. *The backend allows "first-level" malicious attackers to reach well-defined errors, generates no unexpected internal errors, and completely recovers after encountering malformed data.* First-level attacks include anything accessible in request cleartext—including request formatting—but not modification of backend-signed data (i.e., anything following MAC verification).

The second level of checking assumes malicious host code is capable of signing malformed backend data. This level of access, requiring access to the MAC key derived from the controlling WK—i.e., effectively access to the WK itself—is unreasonable in production environments. Since it is logically still an attack on the wire level, we account for this, but acknowledge that we do not expect to encounter it. Second-level attacks are detected by sanity checks, partially assisted by deserialization code of the underlying CSP. Since the backend expects to find properly formatted cleartext after signature verification—generated by another backend, or even itself—malformed data is reported, and the request is rejected. However, such backend error paths are annotated with "SNH" (should-not-happen) marks, to indicate that they are unexpected errors. While these annotations may not be observable in production, our interface test tools verify them in diagnostics builds.

Since production environments are expected to lack direct access to blob-signing keys, *the backend allows "second-level" malicious attackers to reach well-defined errors, and completely recovers after encountering malformed data.* However, unlike first-level attacks, we may mark error return paths reachable through second-level attacks as unexpected (such as reporting them in system logs).

Note that EP11 code includes sanity checks against resource exhaustion and other environmental failures [MKP+95, 5]. Errors encountered during such environmental failures are annotated similar to second-level attacks, with the implicit assumption they "should-not-happen" but are reacted to.

Since object sanity checks may utilize the context of object deserialization, they complement lower-level checks at the infrastructure level, such as referencing untracked memory or NULL pointer checks [PTS+14, 2.1]. Some of the checks inherited from the backend deserialization steps add more context, such as structure size limits, which are then cross-checked against deserialized objects' fields (Fig. 14).

Since the deserialization process provides exact size bounds for all subsequent steps, we may verify exact boundaries for each memory operation—minimizing an important window of vulnerability which is amenable to real-world attack exploitation [AAD+09, 2.1]. Starting from top-level TLV encapsulation, internal consistency checking of each memory region is aware of the allowed number of bytes. This allows us to check memory references within the context of each request, preventing request-originated structures from referencing unrelated memory [Hea14].

While not currently done everywhere, the deserialization step of request parsing with well-defined memory-region bounds could be easily extended to provide "bounded pointers" if additional memory- and type-safety is ever desired. In fact, de/serialization code already manages bounds as out-of-band metadata in a form which may be directly applied to derive bounds for pointers related to request-internal data [Con07, 3.1]. As an example, while the underlying CSP may require use of local pointers within its data structures, those are supplied during import, observing limits of the CSP object size (Fig. 14), de facto preventing the use of arbitrary pointers in most contexts.

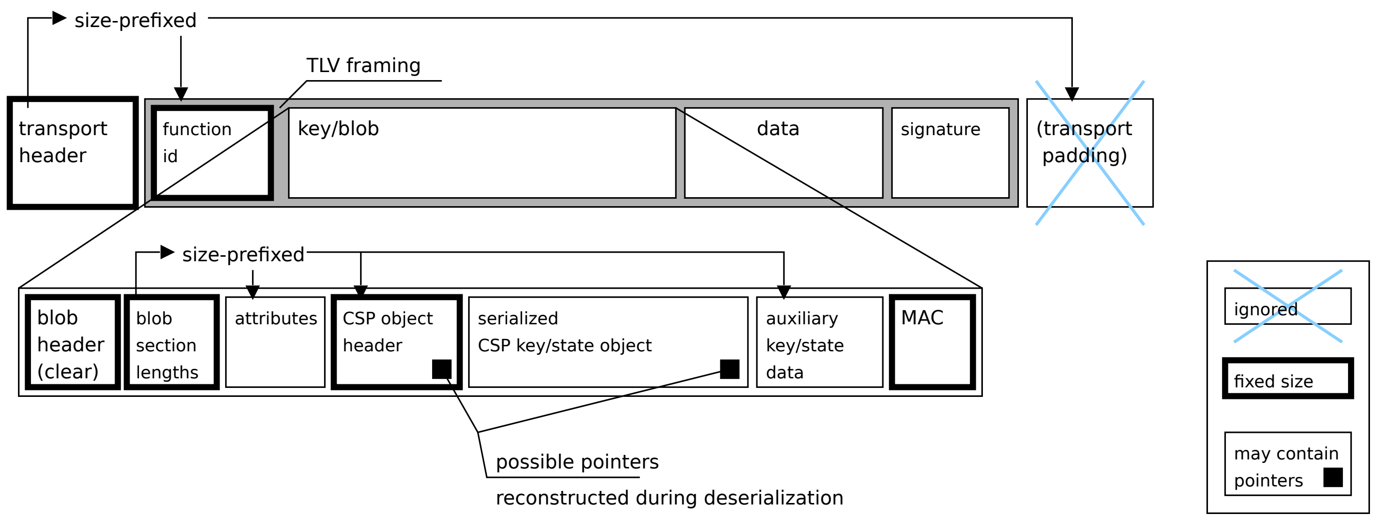

Figure 14: Establishing and tracking bounds on wire-derived memory regions of a VerifySingle request

Malformed persistent data While not strictly an attack interface, our backend must react safely to inconsistencies in persistent data, when reading files within the backend. Our file formats are versioned, and constructed to be future-proof: past fileformat-versions are expected to be recognized. This security assumption is about files where contents' structure does not match the expected formats.

Persistently stored data is integrity-checked: files include a hash, which is verified during each read, but is not visible to code consuming returned bytes (see section 2.5).

We expect to encounter no formatting errors within restored file contents, and detect accidental file corruption of any part of a file—implicitly including failures of any lower-level drivers [PBA$^+$05, 3.1]. If unexpected data is read back after successful file-hash verification, the event is flagged as should-not-happen; the backend recovers from such failures. (While not expected to happen in production builds, these conditions are routinely verified by instrumented test versions, which are capable of injecting errors at these levels. Additional code coverage of such "impossible" paths is a quality metric of our error-injection tools.)

Recovery may include removal of the offending file, and possibly implied removal of others; host code is expected to manage such losses. Fig. 6 shows the interaction of persistent files.

Our regression test suite includes targeted tools to verify recovery from malformed persistent data, essentially observing the backend state machine during reconstruction. These tools are obviously only usable in development enviroments, as production instances lack the necessary access to persistent files.

Data remanence in request-local memory In the stateless execution model of EP11, most request-local memory is used only transiently, and it is released upon responding. (We ignore secondary effects on global data structures, such as cached sensitive state, in this section.) Most transient allocations are stack-based, and are managed automatically, without involving heap-allocation calls. *Our backend minimizes the time where sensitive blob-internal is in the clear* by aggressively wiping any stack structures which could have stored sensitive data.

Our backend code assumes no cooperation from stack management, and we zero-initialize memory unless immediately populated by assignments. *Data structures on the stack, unless demonstrated to contain only wire-visible data, are wiped before their stack frame is released.* Performance considerations prevent us from explicitly clearing *all* stack-resident structures [LSKL16, 3.2], but wire-visible structures may be cleanly delineated—and are clearly highlighed as such in source. (Diagnostics builds include annotations around sensitive stack structures to verify wiping when such structures go out of scope, but we acknowledge that these annotations only demonstrate lack of exposure on annotated paths [LSKL16, 7, Protections using zero-initialization]. However, the effects of these annotations are checked during diagnostics-regression builds.)

Heap used during request processing is initialized in a zero-filled state, i.e., obtained through `calloc(3)`. Heap regions, when released back to the operating system, are wiped just as stack structures are—assisted by the same set of centralized management code. Since dynamic allocations are centralized, only stack objects need to be tracked for initial data, simplifying initialization-related diagnostics tooling [LSKL16, 4.1.1].

While we could special-case code to accommodate operating system modifications adding stack-clearance explicitly [KZ14, 2.4], we chose to implement our own zeroization primitives, and expect no OS cooperation.

Note that our backend supports a minimalistic diagnostics mode, tracking only memory management and the contents of released regions. This lightweight instrumentation allows us to monitor released memory and verify that only zeroized regions are returned for subsequent reuse, even without incurring the overhead of full diagnostics.

9.2.2 Attack scenarios ignored

While acknowledging possible security implications, certain vulnerabilities are tolerated for compatibility reasons, or because they may not compromise our backends.

Our backend does not interact with physical tamper-protection features, assuming selfcontained tamper protection, which requires no reactions from software. We therefore depend on environment-based tamper protection, on a best-effort basis, and acknowledge that we inherit the limitations of environmental tamper protection. See "Physical protection" for assumptions and rationale.

Our backends ignore all certificate parameters except actual public keys. We specifically ignore signatures, expiration or any verification of the signer, relying on certificates only as portable containers for public keys.

Since our administrator identities are effectively public keys, and our backends have only an indirect notion of time—system clock is managed, but is ignored by our administrative interfaces—we may not meaningfully support certificate expiration. Similarly, since we manage whitelists of administrators but never establish trust relationships, or otherwise organize them into hierarchies, certificate verification has no meaning for our backends. Since we rely on a managed certificate whitelist, the entire verification effort is delegated to administrators, including the verification of certificate lifecycles [PKC04, 10.6.2]. Security and trust management through public, append-only whitelists of certificates are practical even for large sets of certificates [AVHS12, 3] [LLK13].

As most of certificates are ignored, and we effectively use only public keys as administrator or KPH identities, checksums indicating ownership and identity are not based on certificate hashes, using key-specific SKIs instead. As a side effect, attempts to register the same public key through multiple certificates will fail, reporting the corresponding SKI to be already used (the practice of issuing multiple certificates is frequent in certain co-hosted/centralized environments [HDWH12, 4.1]).

Hash collisions are acknowledged, but in most cases ignored by our backend, and we do not attempt remediation—other than using SHA-256 as a hash function, or HMAC/SHA-256 as a keyed transformation. Our use of hashes is generally identification, followed by further operations by keys indexed by hashes. The backend rejects incorrectly identified keys: hash collisions may cause false identification, but functional use of mismatched keys is assumed to be detected.

Note that in most cases, comparing by hash is just a prerequisite to use. As an example, even if the truncated form of two WK identifiers collides, the different keys would fail to decrypt blobs encrypted by others. Similarly, if administrator SKIs—i.e., public key hashes—collide, the SKI field within `signedData` structures of such SKIs would match, but the signatures themselves would be rejected.

In practice, with a sufficiently long bitstream from a cryptographic hash function is "assumed-unique" without centralized assignment; also, hashing allows identification without relying on distributed secrets [FKD+13, section 10].

To protect the module against host-induced hash collisions, salt is added to structures where hash-based state is advanced if incorporating host-controlled or known data. As an example, audit events are inserted into a hash chain, and a predefined number of unpredictable bytes will be present within each audit record (9.4.3). *This mandatory salting increases the complexity of attacks attempting to control hash-chain state (evolution).*

CBC unpadding using PKCS padding—possibly other padding modes, if they are added in the future—may serve as an unpadding oracle, revealing plaintext contents by returning plaintext-dependent error codes during decryption [CHVV03, FP13].

While side channels caused by decryption are in fact practical, we implement the PKCS#11 standard properly, and are therefore obliged to return such selective error values. Also, since we do not separately access-control unpadded CBC mode—which is a prerequisite of its padded relatives—the entire plaintext may be obtained at the cost of an extra call. Therefore, we delegate the problem of masking—actually, obfuscating—plaintext-related errors to host code.

Similar decryption/validation oracles, when they are implicit in protocols, MUST be accommodated in protocol-aware code [ASS+16, 3.1]. The lack of granularity of PKCS#11 calls prevents us from recognizing how, as an example, decrypted blocks are processed.

Attribute-bound keys are imported without verifying key checksums in the header even if all participating key checksums are present—i.e., those of the key, the KEK encrypting it, and the MAC/signature key. These checksums are constructed properly at export, and are ignored during import.

Note that the MAC/signature key authenticating the AB-wrapped key may easily replace any checksum within the header, then construct a valid signature over the updated compound. Against any other attacker, including accidental corruption, the MAC/digital signature already protects the full key. Therefore, not verifying the checksum fields introduces no new vulnerability.

Audit sequence numbers may wrap, which is acknowledged but not separately accounted for. In a practical setup, unless administrators maliciously update clocks, the sequence number rollover would be noticeable. (Even after a malicious time update, the aggregate hash chain would obviously link audit history.) Due to these restrictions, we do not protect against a sequence number wrap, other than issuing a special-purpose event to mark it.

In practice, we use a sufficiently wide counter—considerably over 32 bits, see the wire section—to be able to consider audit-counter wrapping impractical.

Audit records are delivered asynchronously to the host, therefore *creating a time window between delivering results and the arrival of the corresponding audit records* (see the functional-call flows in Fig. 16). If a module is restarted, or otherwise becomes unreachable during this period, the system will lack the audit entry for the referenced event. This asynchronous window is inherent in the operation of our audit chain, and we acknowledge it without attempting to fix. Systems with high-assurance requirements may mandate some kind of redundant audit-chain storage within disjoint security domains on the host to prevent malicious denial of the most recent entries [CW09, 2.2] [BMC+15, VIII.A.]. Alternatively, one may procedurally require confirmations through multi-stage commits, a common technique used in asynchronous hashchain-based transaction systems [Nak09, 8] [cd14b].

Since we made the conscious decision of not adding audit data to responses directly—i.e., keep our interfaces similar to PKCS#11—there is an inherent asynchronicity between multiple events. Note that similar problems are prevalent in any remote, inherently asynchronous interface; they are highlighted since audit events are worth of special mention.

To simplify the synchronization of functional and audit-record state, the backend includes both unsigned and administrative-signed queries to retrieve the last audit events from a circular buffer of reasonable capacity (Fig. 16, see section 9.4.5). Therefore, *high-assurance applications requiring strict audited proofs* are encouraged to wrap audit-relevant operations, couple them with immediately following audit queries, and only return confirmed objects when their audit event has been received. Substantially similar algorithms are used in distributed payment systems to create confirmed transactions [KAC12, 3.1], and we expect host libraries to wrap similar functionality if needed.

Note that updates of the audit state within the module are atomic: when audit-event construction is reported complete, the persistent copy of the audit state has already been committed to disk. The loss of audit record only refers to the newly generated audit event; the event chain is expected to advance in an atomic fashion regardless of when the backend is reset.

Note that partial protection may be provided against event chain truncation, if state-delimiting "metronome entries" are inserted into the event chain, for example by issuing audit-relevant requests. Using such extensions, the premature termination of the event chain becomes immediately visible, and the frequency of metronome entries also provides secondary information about missing time windows (duration) as well [Hol06, 8].

Audit records may not be reliably generated during very initial stages of system startup, therefore these early events may be logged only approximately. As an example, audit state may be restored from persistent storage only after a successful KAT of hashing, since it depends on verifying the hash of the state file first. Therefore, audit-relevant events related to known-answer testing—of at least hashing—or initial filesystem access may be generated before audit startup. Our workaround for such initial events is an offline audit-event queue, which preserves only the chronological order of events, but does not provide details beyond a single—possibly quite specific—reason for each event. As most early events are actually quite concise, such simple early-logging queuing is satisfactory.

Due to its intentionally simplistic design, *early-event logging has two serious limitations: only the chronological order of events is preserved, and the backend only offers a finite capacity queue for preceding events.* The latter limitation is not externally visible, since finite capacity is oversized relative to the worst-case capacity required by EP11 itself, but it is mentioned here for completeness. (Internal headers and specifications unambiguously describe any limitations relevant to EP11 developers, but this information is not relevant or made visible to the host in practice.)

When the audit subsystem has started up successfully, it immediately processes any pending early events. In practice, successful startup is quite close to powerup: delays are expected to be comparable to typical device-access latency in high-end servers; in any case, in the range of event-timestamp resolution. We consider the effect of inaccuracy in such early-event records negligible. Therefore we acknowledge that early event timestamps may be slightly inaccurate—but their relative order will be preserved—and do not consider this inaccuracy to be security-relevant.

Note that EP11 in fact preserves event identifiers to mark the start of early vs. regularly submitted audit events, but currently we do not—yet—issue such separator events in production.

Incremental decryption of streams may reveal information about plaintext, even if the full stream would have been rejected due to some integrity mechanism failing. As an example, earlier parts of an improperly padded CBC-encrypted stream may still be incrementally decrypted, and the last call would fail due to the detected plaintext being invalid.

This vulnerability is inherent in incremental calls; it may only be prevented by mandating single-pass processing. As PKCS#11 provides no standard method of restricting symmetric processing to single-call operations, we acknowledge the vulnerability, but do not protect against it.

Note that the inherent problems of incremental decryption returning plaintext before final verification are widely recognized [Lan14]. For standards where this may become a problem, the recommended solution is to delegate final reassembly to outside the cryptographic primitive, if the environment may not tolerate partial decryption [IEE07, 4.4.2].

The certificate chain of imported state is not verified in its entirety, even if it is included in system state (8.3.2, Fig 13). The signature covering the file is still verified, but the certificate chain leading to the signing key is not. Note that the import procedure reassembling the import file requires administrative commands (Table 2), so administrative approval on the state is implicit. The signature on the file itself is basically an integrity check on contents, but contributes no additional security.

Note that our HSMs are issued certificate chains originating at the factory CAs, but certificates of the latter are not present in the modules themselves. Therefore, the certificate chain in an imported state terminates is effectively self-signed: administrators authenticated the whole chain, but we may not verify the root certificate through an independent channel. Under such circumstances, verifying the file signature may detect corruption, but provides no security. *If malicious administrators collude to import an invalid state file, they could also conjure a similarly "self-signed" certificate chain to authenticate it.*

Note that the lack of verification is not a security problem: we generally protect *export* of secrets, but importing essentially arbitrary data—as long as administrators approve it—may not compromise other backends.

Obviously, regardless of the signature, data formatting and consistency are verified during import. Since the presence of a file signature does not bypass such sanity checks, backend integrity may not be compromised even by maliciously constructed, properly signed imported state.

Properly signed blobs containing invalid keys may produce invalid results, which the backend may not detect. Note that under reasonable conditions this should not happen, as we assume WKs would be procedurally protected during normal operations. Note that anyone controlling a WK may inject arbitrary content into a blob. Therefore, our goal is limited to proper backend operation even in the presence of such malformed—but properly MACed—blobs.

Note that after the signature on MACed contents is verified, the backend performs all applicable format checks, and will reject structurally corrupt blobs. Assuming MACed contents were valid, subsequent structural errors encountered are marked as unexpected—should not happen, "SNH". In fact, maliciously MACed blobs are actively used by our test tools to check for these conditions during development. However, due to lack of redundancy most objects types may not be detected as "corrupted" if they are properly MACed. As an example, secret keys of state-of-the-art symmetric algorithms lack any structure, and raw key material is expected to be statistically random [RS06, 1]. Therefore, modification of only the raw key bits may not be easily detected, a known key-storage artifact [ST16, 5.1]. (Even if we added separate integrity fields to raw keys, a malicious WK owner could also recreate those, matching any corrupted raw key bytes.)

Due to the limitations on integrity-checking raw key material, and the inherent difficulty of validating some keys, we justifiably limit our checks to object formatting which may influence backend control flow. We therefore acknowledge that a malicious WK owner may change raw key material, and such replacements may not be detected, and not even attempted to do so, by the backend.

Note that *certain unexpected internal errors may expose the timing of internal checks, if encountering errors later than successful verification of properly signed data.* These conditions are assumed to be reachable with a non-negligible probability only if the attacker already possesses the proper signing/MAC key, and then additional timing information has no further value to attackers. Therefore, *timing behaviour of such unexpected errors is approximately constant, but only on a best-effort basis.*

Diffie-Hellman keys may leak information about their secret-key bitcount with a low, non-zero probability when unwrapped. Since we need to reconstruct the standard PKCS#11 parameter describing the bitcount of the `X` DSA parameter (`CKA_PRIME_BITS`), `X` is retrieved from the wrapped private key, then rounded up to 32 bits. *When the generated `X` value has* *more than 32 leading zeroes, we reveal this condition as a side-channel* by reporting the short X (this happens with a probability of 2^{-32}). Practically, while the intended bitcount is an upper limit, we reveal if the actual X value is considerably shorter.

Since the presence of PKCS#11 parameter is mandated by the standard, and regular PKCS#11 wrapping modes lack sufficient metadata, we may not protect against this weakness without breaking standards-compliance. We may add a control point to control rounding—sacrificing strict compliance for security—as a future extension. (Reporting inexact, conservative PKCS#11 parameters should remain within the standard [PKC04, 11.2]).

Note that our proprietary attribute-bound format does not have this problem, as the originating bitcount may be included in our wrapped form along with other attributes. The vulnerability is unique to standard PKCS#11 transport modes.

The backend performs per-request usage-restriction checking at well-defined points, and does not react to subsequent changes to restrictions, even if they happen during execution of that request. As an example, if the CP setup allows an operation, it is allowed to complete even if a simulatenously submitted administrative command disables a relevant CP before request completion.

We acknowledge that (almost-)simultaneous functional calls and administrative ones changing restrictions interact, but observing the overall asynchronous nature of a multithreaded backend, we accept this condition. In general, *we only state that restrictions are evaluated in a manner consistent at the time of checking.* Since usage-restrictions are checked at multiple levels, our statement is valid for each of the potentially multiple cross-checks; see section 2.6.

Note that real-world scenarios tend to procedurally restrict administration to otherwise quiesced modules, even if these are not taken completely offline, just temporarily bypassed by functional-request routing. The primary reason for temporary quiescemence clear is separation of before-after states around administrative changes; this, coincidentally, closes the window where our approach may be criticized.

Reliable auditing requires host storage and procedural cooperation to store audit logs.

Note that all HSM-resident systems must provide positive proofs, since the modules may intentionally self-destruct as a regular, expected operation—such as during a perceived tamper attempt. Therefore, mandating positive proofs and the necessary host procedural controls is not unique to our backends, and should be expected in any high-assurance environment, or any application aware of HSM operational details.

Administrators are authorized to remove logged-in sessions even if they may not reconstruct the originating PIN/nonce combination. This capability is needed since sessions may remain logged in to any module, impossible to remove, if the session identifier and the originating PIN/nonce are no longer available. Since administrators authorized to issue the session-removal command would also be capable of zeroization, allowing removal of a single session is only a subset of already available functionality.

Note that while forced session removal is available, administrators may not log in a session without the original PIN/nonce. *Therefore, administrators may invalidate a session, but are not capable of impersonating the originating user.*

We derive session identifiers with a fixed key calculating HMAC/SHA-256 over host-provided data (see 2.3), and do not—currently—support diversification through selecting alternate keys. We effectively treat our session-deriving key as a non-sensitive shared value.

We acknowledge that adversaries capable of constructing HMAC collisions may impersonate other entities by recreating their session identifiers, and do not intend to fix this vulnerability.

Since host administrators are already assumed to be able to observe host-provided login traffic, they do not even need to construct HMAC collisions to impersonate other host entities. Against non-privileged host-based attackers, we assume that enumerating login information is simpler than attacking the HMAC algorithm itself.

Note that our stateless model, and the fact that all modules may recreate sessions, prevent us from meaningfully restricting login events through rate limitation.

We acknowledge that the derivation of identifiers does not offer security comparable to that of MAC calculations. Since sessions must remain logically separate from WKs, session-identifier derivation must behave identically on all backends, even in the absence of synchronized WKs. Therefore, relying on a single, fixed key to diversify an otherwise standard HMAC calculation is the only feasible solution, and it is considered an acceptable tradeoff. Note that practical "secure" protocols are de facto deployed with similarly shared keys [GHC14, V.D]—but unlike our modules, those "private" keys should in fact be sensitive.

Single-binary instances of EP11 are not protected against malformed requests or compromise through persistent data. Configurations where host and backend code execute in the same address space are assumed to be under complete control of those capable of issuing requests. *Our security assumptions about protecting against malformed requests or persistent data may not be enforced in such environments.* We acknowledge this restriction, and ignore it, since no production deployment operates with identical host and backend address spaces.

Note that single-binary test instances of EP11, or backends accessible through a debugger, are the most relevant instances which are vulnerable through address-space sharing. These instances should not manage sensitive production key material, and the capability of directly modifying backend state is not expected to add new vulnerabilities.

We depend on compilers not bypassing defensive code, such as bypassing memory-wiping calls, or otherwise disrupting features added to increase resiliency. This dependency includes certain countermeasures, such as compiler optimizations discarding certain checks as "impossible" conditions [WZKSL13, 2.2]. We assume proper compiler operations under these conditions, and may need to audit generated code for proper operations.

Note that many of the defensive features—such as memory comparison with guaranteed data-independent behaviour—use only standard code structures, and are therefore not subject to potentially erroneous compilation. Unfortunately, these structures tend to be inefficient, but we tolerate the performance penalty to preserve portability.

Certain "attacks" on the entropy source are acknowledged but may not be defended against, if the entropy source is malicious. As an example, even a trivial amount of state and code may generate "normal numbers" which would be, within reasonable bounds, indistinguishable from a perfect source—at least, given the limitations of our module-internal checking [Bai04, 4]. *Since our capability to detect a malicious source is limited, we only attempt to defend against realistic failure modes of a non-malicous raw-entropy source within the entropy conditioner* (TRNG, "true-random" generator).

Obviously, within the tamper-protected boundary of an HSM, physical replacement/bypass of the source is infeasible. Similarly, we assume connections between the source and the TRNG conditioner are physically protected. Therefore, we may legitimately assume that only the source itself may be compromised/hostile, but attackers—other than those who have compromised the source before its integration—SHOULD be unable to manipulate the raw-entropy stream.

See section 9.3.1 for the limitations, the full rationale of entropy estimation, and the threats considered by entropy conditioning.

9.3 Random-number generation

The backend includes a "hybrid" random-number generator, seeded by *conditioned entropy* which conditions the output of a dedicated physical source to compensate for relevant statistical deficiencies if possible [KS11, par310]. The conditioned entropy in turn is passed through deterministic post-processing to generate application-visible output [KS11, 4.5.1] [BK12a, A.4] (Fig. 15).

The dataflow between the raw-entropy source and application-accessible random bytes may be equally interpreted as a monolithic, hybrid DRNG incorporating hardware-originating entropy, or as a high-quality conditioned entropy source feeding logically separate DRNGs, depending on whether the logical separation is considered relevant ([KS11, 4.9] or [KS11, 4.5, 4.8], respectively). See [KS11, par310] for an example why such structural decomposition may be subject to interpretation.

Applications do not directly observe the raw entropy source, and—since DRNG structures are separated from conditioned entropy—multiple instances of the deterministic generator may independently coexist.

The *conditioned entropy source periodically reseeds one or more instances of deterministic generators (DRNG),* using *full entropy* extracted from a potentially non-ideal "raw" entropy source [KS11, par302] through a cryptographic one-way function [KS11, par304, par309, 4.8] [BK12c, 6.4.2.2].

The following description references standards relevant to random-number generation for high-assurance applications [KS11, BK12c, BK12b, BK12a], referencing more than one if applicable. Many specific references to [KS11] include paragraph numbers.

9.3.1 Entropy source

Our system includes a single, centralized entropy-extraction component, potentially shared between multiple, independently seeded DRNGs. This *true-random generator* (TRNG) entropy source is "conditioned," *designed to generate full entropy* output from a dedicated hardware source [KS11, par302] [BK12c, 6.4.2], even if the source may not be statistically ideal. The TRNG aggregates past history of the raw-entropy stream through "folding" it into a pool of fixed size, XOR-ing newly arrived bits forming a circular buffer (see a similar description in [KS11, par316]). The pool is intentionally oversized to allow full-entropy accumulation even if the raw entropy feeding it has a low, but non-zero, real entropy rate. When entropy is requested, and

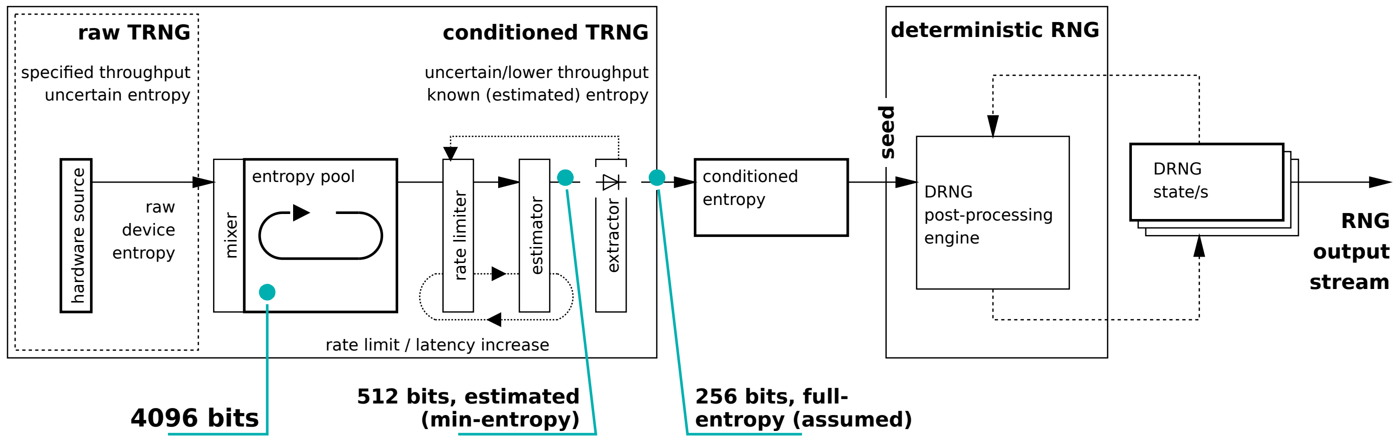

Figure 15: Random-number generation, overview

conditions allow, *pool contents are passed through a cryptographic hash function,* producing the next conditioned-entropy block.

In terms of [KS11] `PTG.3` functionality, pooled raw entropy forms an "inner" PTRNG [KS11, par309, PTG.2], and extracted entropy corresponds to the "internal" state of the *TRNG-internal* DRNG postprocessing according to [KS11, par310]. Note that the DRNG here specifically refers to post-processing—conditioning—within the `PTG.3` source itself (Fig. 4 in [KS11, par263]), and *this mention of DRNG is logically separate from subsequent deterministic post-processing.*

Entropy extraction is gated by a rate-limiter and an entropy estimator step before extraction is allowed (Fig. 15). The rate limiter simply prevents extraction before at least $M = 512$ non-zero raw-entropy *bytes* have been mixed into the pool (in practice, considerably more than 512 bits will have been mixed as part of such an 512-byte raw-entropy stream). The rate limiter therefore ensures that $M \geq 2N$ new bits have entered the state before $N = 256$ bits of conditioned entropy are produced [KS11, par316] [BK12c, 6.4.1]. Blocking until a "sufficiently high" number of new entropy bits has been mixed in prevents iterative guessing of pool state through small, consecutive, incremental reads [FS03, 10.2] [KSF99, 3.2] [Str16, 3.1]

Entropy estimation of pool state uses a min-entropy estimator, providing a conservative estimate of pooled-entropy quality [KS11, par332] [BK12c, 4.2]. Note that the the pool size is several times the size of required apparent entropy, allowing proper operation even when backed by a significantly biased—or otherwise non-ideal—raw-entropy source. Long-term, offline statistical evaluation of a sizeable group of discrete raw-entropy sources used by IBM HSMs show them to be *independent and identically distributed* (IID) in their steady state [BK12c, 9.1.1]—a statement reinforced by component documentation.

Once permitted by entropy estimation and the rate limiter, entropy is extracted by computing a cryptographic hash of the entire pool contents. This turns a "fresh" pool state containing at least 512 bits of apparent min-entropy to a $N = 256$-bit conditioned-entropy block through SHA-256 [Nat12, 4.1.2]. Since this step combines a hash function with a conservative entropy estimate reporting more than $2N$ bits of apparent entropy for N bits of output, *entropy extraction may be assumed to produce blocks of full entropy* [BK12c, 6.4.2.2]. Entropy extraction reduces the rate of conditioned entropy compared to raw entropy input considerably [KS11, PTG.3.6]. Since we use a cryptographic hash function as an extractor, conditioned TRNG output is assumed to pass all reasonable statistical tests [KS11, PTG.3.7]—a design goal of all hash functions.

TRNG failure scenarios are handled by a combination of rate-limiter and min-entropy estimator:

- Loss of the raw-entropy source—such as a "stuck-at" fault preventing generation of new bits—prevents subsequent compression as the rate limiter rejects subsequent requests to compress [KS11, PTG.3.1].
- Since stuck sources are already detected by the rate limiter, no subsequent statistical testing is required to check for this failure mode [BK12c, 6.5.1.2.1].
- The min-entropy estimator is particularly sensitive to repeated values, if their relative frequency within the pool crosses a threshold. Therefore, as a side effect, our conditioning will after some delay recognize repeating values [BK12c, 6.5.1.1] without targeted checking. While generally not expected from a dedicated entropy source, raw samples from a quantized, smaller-than-ideal set would be similarly detected if the raw source is deficient [Dic15, 4.2].
- Long-term statistical deficiencies of the raw-entropy source would be partially compensated by XOR-ing non-overlapping streams of raw entropy into a single pool, then estimating min-entropy before extraction [KS11, par290]. This continuous entropy estimation effectively forms an automatic, online test procedure [KS11, PTG.3.5] [BK12c, 6.5.1.2].

When accumulated long-term deficiencies lower min-entropy below the acceptable threshold, extraction is stopped instantaneously [KS11, PTG.3.4] [BK12c, 6.5.1.1].

- *Structural failures repeating the same 256-bit conditioned entropy block in adjacent blocks are reported as a catastrophic TRNG failure,* and this stops execution.

  Note that even an ideal TRNG would also, with a negligible probability, generate identical adjacent 256-bit blocks. This possibility of false positives is acknowledged, but due to the extremely low probability, we do not intend to remedy this. Note that our false-positive rate is orders of magnitude below what is considered tolerable for real TRNGs [KS01, Class K2, (e)].

9.3.2 Deterministic random-number generation

The pseudo-random generator is based on a non-invertible, cryptographic hash function (SHA-256), instantiating the DRBG structure from [ISO11, C.2.1.1], which itself conforms to DRG.3 requirements [KS11, 4.9.1] as described in Example 39 of [KS11, 5.6.2]. The DRNG is seeded by maximum-length entropy—256-bit blocks—from internal, conditioned seed, and maintains state with up to 256 bits of entropy. As mentioned before, whether the DRNG is considered separately from the entropy conditioner, or the entire dataflow is treated as a DRNG incorporating entropy, is functionally immaterial.

Instantiated as a single hybrid RNG [KS11, 4.2], different callers within the TOE obtain their own slices of the generated DRNG stream. TRNG-based reseeding is automatic, and it is not influenced by external entities in the TOE configuration. The rate of reseeding is controlled directly at the DRNG level, forcing entropy extraction and reseeding after a predefined number of bytes have been extracted from the DRNG.

9.4 Auditing

The backend provides a high-assurance audit log based on hash chains, preserving past history through a non-malleable, unambiguous sequence of audit records. Records are generated internally by the backend, are output through an asynchronous mechanism—such as `syslog`—and the last entries are simultaneously made available through synchronous queries (Fig. 16).

The audit infrastructure provides a transparent, high-assurance mechanism to monitor the evolution of system state. Aggregated through a cryptographic hash function, the audit chain is expected to be published without restrictions, and must be available for processing without cooperation of the originating module [LLK13, 1] [MA14, 3]. As an additional administrative query, audit records may be exported in a module-signed structure, if high-assurance integrity proofs are required. With the addition of timestamp-specific keys, the current infrastructure may be trivially extended to sign collections of events, grouped hierarchically, using RFC-standardized formats [GBP07, 4.2].

The audit system state is described through a single hash per backend, aggregating past history into a single, non-malleable sequence of hashes. *Since records are inserted into a hash chain, the entire audit log may be stored on untrusted hosts, and the module-internal persistent state has a modest, fixed size.* Audit records may be unambiguously ordered if issued between module zeroize calls, potentially allowing reconstruction of the entire audit chain even from an unordered, potentially redundant, but complete set of audit records.

While audit records are serialized and chained in a single stream—per module—any particular entry may be obtained as a module-signed response, and therefore may be verified on its own. Since this capability is available, we assume hosts would be able to store chains in their entirety, or obtain signed versions of specific states and use them to "skip" sections of the entire history [CW09, 3.1]. Note that our modules lack global context of audit chains, and therefore may not construct hierarchical "history trees" as described in [CW09, 3]. However, our capability to authenticate any required state still allows us to partition the module-global history into smaller, individually verifiable sub-sequences.

We expect enterprise systems to integrate audit chains from multiple backends, and maintain them in a global hash chain, allowing global synchronization and audit event verification. As deployments of the `Bitcoin` global hash chain indicate, practical hash chains of considerable complexity may be easily parsed in their entirety on state-of-the-art hardware [SMZ14, 3.3] [RS12, 3].

Among other things, the following security-relevant entries are always included in the event chain:

1. Key generation
2. Key import (i.e., PKCS#11 `UnwrapKey`)
3. Key export (PKCS#11 `WrapKey`)
4. Key destruction (note that this is only possible for SRKs)

5. Administrative commands changing state, including module or domain state import or export
6. Changing module time, as a specific state-changing administrative command
7. Startup and selftests of the backend and relevant subsystems.

Audit entries unambiguously identify the originating module, the then-current clock, a monotonously increasing sequence number, and initial/final audit states. *This level of detail allows unambiguous reconstruction—and integrity verification—of the event chain from a collection of audit records.*

9.4.1 Audit record construction

9.4.2 Audit record, fixed content

Audit records combine fixed header fields to provide general context, including at least these fields:

1. Sequence number, a monotonously increasing counter (restarted at zero after module zeroization)
2. Event time, with millisecond resolution
3. Initial audit state (hash)
4. Final audit state (hash)
5. Module identification: serial number and instance
6. Event type and invoking function
7. Originating domain, if applicable

As with administration, auditing includes a randomly generated instance identifier, separating multiple audit chains of the same module. This additional identifier separates multiple audit chains of the same module, even if zeroization resets the sequence number, and time is modified to recreate identical times.

Audit instance, together with the sequence number and current state (hash), is maintained in a persistent file, outside administrative control (Fig. 6). This file is preserved across any card-zeroize event, to ensure auditing is consistent across the entire lifecycle of the card. If the contents of audit structures are ever corrupted, a new audit-event root is generated, containing the corruption and recovery as the first recorded event.

9.4.3 Audit record, variable content

In addition to fixed content, a set of optional fields is defined for records, each relevant only to a subset of auditable events. One of the header fields describes the presence or absence of optional fields, therefore record decoding is unambiguous.

Optional fields are the following:

1. WK identifier
2. final WK identifier, if the event includes a WK change
3. compliance status of the hosting card/domain
4. key record or records, of all participating keys

   There are up to three keys used by all operations: base key, key-encrypting key (KEK), or integrity (MAC) key. Exact assignment is documented under each functionality group; keys present are assigned in this order.
5. compliance of newly generated, imported, or modified keys, if different from current one
6. salt, with a minimum number of module-generated random bytes.

   Salt is inserted to prevent the host from issuing audit records with mainly host-influenced content, therefore preventing the host from controlling audit state (i.e., hash).

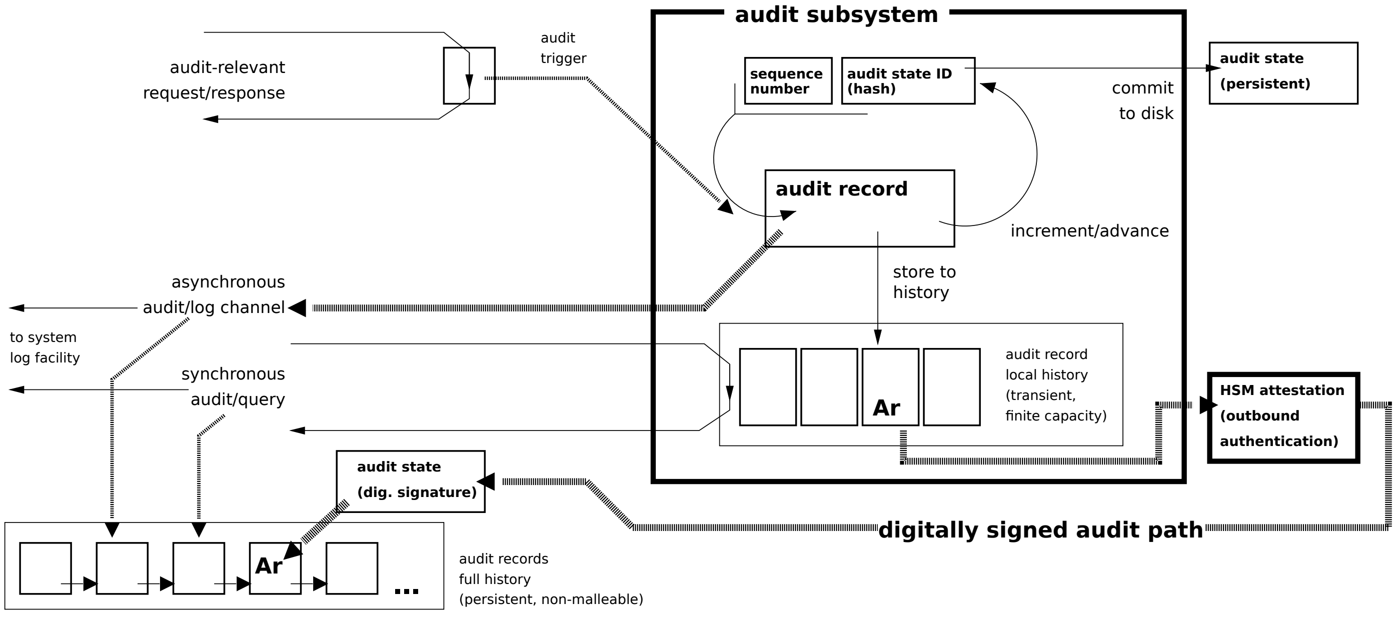
audit subsystem
audit-relevant request/response
audit trigger
sequence number
audit state ID (hash)
audit state (persistent)
commit to disk
audit record
increment/advance
store to history
asynchronous audit/log channel
to system log facility
synchronous audit/query
audit record local history (transient, finite capacity)
Ar
HSM attestation (outbound authentication)
audit state (dig. signature)
digitally signed audit path
Ar
...
audit records full history (persistent, non-malleable)

Figure 16: System audit: control and data flow

7. deterministic, salt-like pseudo-random function (PRF) output, providing a simple in-band "checksum" linking record-identifying fields to a PRF. This short field of high "apparent entropy" is fully determined by event context, and may be verified within the record.

   The PRF-derived field is not expected to contribute to cryptographic strength; it is, however, included in the event-record payload (therefore, it is covered by the hash chain). The operational value of a context-dependent, PRF field is potential disruption of hash-collision attacks, by restricting the solutions of hash-collision constructions to those where PRF output and its base fields are consistent [SBK+17, 5.5]

   Note that the goal of deterministic salting differs from that of salt fields. The latter prevent malicious host entities from advancing the hash state in a fully deterministic fashion, but may not prevent against later forgeries—since the attacker MAY modify the value of salt fields when attacking an already issued event record. Deterministic, salt-like PRF fields MUST be consistent even in forged event records, therefore their contribution is against later, offline forgeries.

The number of salt bytes varies, depending on the nature of audited event. For events which include newly generated keys, object MACs, or other newly generated uniform-random bytes, the number of salt bytes may be reduced.

Audit record, key records Details logged about each participating key during audited events include the following:

- Key type
- Key size, encoded in a type-specific way, logging relevant sizes (such as both P and Q bytecount for DSA keys)
- Key identifier, such as PKCS#11 checksum or other control value
- Controlling session, corresponding to the identity of key owner
- parts of the MAC of newly generated objects

Object MACs are stored in a truncated form, allowing matching of specific objects to events, even if the same key appears in multiple tokens. Since MACed objects are all randomized, identical keys in multiple blobs will be logged with identical key identifiers and different object MACs.

Other details, such as domain or controlling WK are included in the record header and are not replicated within key records.

9.4.4 Audit security

Audit entry disambiguation Audit entries include only hashed/truncated forms of identifying fields such as keys or identifiers. These mappings are, by construction, not bijective, and keys/objects/sessions may not be unambiguously reconstructed from audit events. *The intended use of audit entries, unambiguous association of a specific key/object/session with a specific audit event, is obviously possible.* In a typical case, a high-assurance application could also store audit entries corresponding to security-relevant events, falling back to inspection of the audit chain if later required.

We acknowledge that there is potential ambiguity in associating identifiers with truncated audited forms, with collisions of 32-bit or larger hashes as false positives. Conversely, it is never ambiguous if an audited identifier differs from a full one; *our audit infrastructure is designed to assist with such object identification*. Note that being able to demonstrate only a negative—i.e., different truncated identifier/hashes must belong to different objects—is not different from hash functions.

Note that *several of the audited identifiers,* such as session identifiers, are used in proof-of-possession schemes, and therefore *must not be logged in their entirety.* This restriction—in addition to size constraints on audit events—forces us to use a potentially ambiguous, truncated audit representation.

9.4.5 Audit history

In addition to logging audit records through regular log facilities of the host, the audit subsystem retains several of the last audit entries in a transient, in-memory table. A non-administrative query is provided to output table entries; note that past entries already discarded from the table are no longer accessible within the module.

Since audit history queries are returned as regular—synchronous—responses, they may be used if the unpredictable asynchronous latency of system logging is problematic. In high-assurance cases, synchronous queries should be used to obtain audit entries with a more controlled latency.

As an alternative for high-assurance environments, an administrative query is also available, if the audit state needs to be authenticated—i.e., OA-signed—by the backend. Since the hash chain already provides non-malleability, the administrative query is only recommended when the audit state requires more persistent authentication, such as attesting specific states for long-term persistent records.

Long-term archival of audit history is expected to include known-good hash chain states with auditors approving history up to each checkpoint. In such cases, publishing a list of known-good states allows users to check chain integrity incrementally, using only recently generated audit records [cd14a] [LLK13, 8]. Assuming advancing chain states to specific values, history may need to be retained only since the last checkpoint, a common recommendation for hash-based systems which deal with potentially large hashchain-based history [Nak09, 7].

Audit persistence Since the audit chain may be securely stored outside modules, the internal audit-chain state is stored in a minimalistic, integrity-checked structure, containing only the following fields:

1. Audit state (i.e., hash)
2. Sequence number/counter
3. Audit instance

This set of fields is sufficient to reliably concatenate audit chains internally. All other fields are constructed at audit-event creation, and need not be retained in persistent storage.

As with other module-internal files, the audit state is integrity-checked through an embedded, transparently generated and verified cryptographic hash (see 2.5). Since recovery from files requires verification of the hash infrastructure first, the first stages of system startup can not be reliably audited. The security rationale of the recovery process is described on page 9.2.2. Note that the embedded hash—not observable outside the module—of the audit-state file is disjoint from any audit state hash.

9.5 Miscellaneous security-related notes

9.5.1 List of algorithm selftests

Module startup, or the query-triggered algorithm selftest verifies known-answer tests of the following algorithms, depending on the underlying crypto provider/engine supporting them:

1. hashing: SHA-1, SHA-2 [-224, -256, -384, -512, -512/224, -512/256] [Nat12]
2. HMAC: SHA-1, SHA-2 [-224, -256, -384, -512, -512/224, -512/256] SHA-3 [-224, -256, -384, -512] [FIP02]
3. AES encrypt/decrypt, 128, 192, 256-bit keys, ECB/CBC modes [Nat01, BB12]

4. TDES encrypt/decrypt, 128, 192, 256-bit keys, ECB/CBC modes [BB12]
5. CMAC: AES, 128, 192 or 256-bit keys, CMAC generate and verify [Dwo05]
6. RSA: sign/verify, with PKCS#1/SHA-256 or PSS/SHA-256 padding
7. RSA: encrypt/decrypt with PKCS#1 or OAEP padding
8. ECDSA: verify KATs; sign/verify cycles (all NIST P curves; BP-192R)
9. DSA: verify KAT; sign/verify cycle (2048/256-bit P, Q)
10. DH: KAT between known keypairs

Algorithms prohibited by policy restrictions, i.e., not accessible through functional API calls, MAY still be tested—such as when considered relevant to verify engine connectivity. (As an example, 112-bit TDES may still be tested, even if policy settings prohibit use of this mode.)

9.5.2 Prohibiting instance identifier export

As documented under the details of state export, the "full" state export does not include module, domain, or audit instance identifiers. Any module importing full states would generate new identifiers internally.

Prohibiting export of identifiers is intentional, not an oversight. Exporting these identifiers could enable replay attacks if the exported state is subsequently imported back to the same module (which would have, obviously, the same serial number, number of domains etc.).

Note that a test extension exists which saves or restores all identifiers prohibited from export, as one of the diagnostics-only additions. As with other extensions, this feature is not available in production builds.

9.5.3 Use of external seeds

When backends allow external seeding, which may be disallowed completely, or during runtime by CP setup, any externally provided seed is mixed to the module-internal one, never replacing it. Therefore, the entropy of seed should not fall beneath that of the higher of external and module-internal source.

Note that test features capable of exporting or importing RNG state are obviously exempt from restrictions on seeding. Since production builds do not contain these test features, allowing full export if test features are enabled is obviously not a security problem.

9.5.4 Test types

Diagnostics build add test functions, dedicated to test-related functionality, which (the "`SYS_TEST`" collection). *None of these, potentially insecure test functions are available in production builds,* but they may be enabled with a single setting, not changing other, production-included functionality.

Set WKs, both current and next ones, including optional commit operations, and combinations related to imprinting (such as: set a WK and imprint its domain, to enable functional testing with a single call)

Set CPs in any domain

Zeroize the module or any of the domains

Encrypt and decrypt data with different algorithms in loops, with host-supplied iteration counts, for symmetric and RSA en/decryption modes.

These looped en/decryption functions are made available to simplify side-channel analysis. They are functionally identical to submitting the same requests from host loops, but they maximize

Scalar multiply constants on specified EC curves, in a loop. This extension may be used to test EC implementations for side channels. It provides closer access to the actual scalar multiplication than practical deployments do, so testing through this function is a conservative overestimation of realistic attack scenarios.

Generate RSA keys from externally supplied prime seeds

Generate DSA parameters from specific seeds and iteration counts. This functionality is a direct mapping from NIST tests of DSA PQG generation.

Read or write arbitrary temporary files within the module. While a regular administrative query and command are available, the test-only function allows us to process files without signatures (file transport involves many signatures, due to file fragmentation caused by large files and small per-request data chunks).

List administrators or SKIs are similarly mirroring regular administrative queries, but may accelerate testing since they do not require signatures.

Admin-sign arbitrary data may be used to test error-recovery of the backend, when malformed but properly signed data is submitted to it.

Query backend CSP configuration including any details intentionally not reported through regular queries. This functionality is mainly useful for testing where details of CSP headers are useful, such as to maximize the coverage of our fuzz-tester, or other tools which benefit from undocumented insider access to the backend.

Query or set filesystem read-only mode as described in section 2.4.

Save or restore the module-internal RNG, including its TRNG seed. This operation allows us to reconstruct most non-deterministic operations purely based on host commands.

Query or modify performance-measurement settings, including activating different modes and precision

See the wire section of the full list of functions and any rules on parameters.

10 The EP11 host library

The EP11 Host Library is the C reference implementation of the wire protocol (see p. 130 onwards) and is part of the IBM Z EP11 Support Program. The information here is included for reference purpose only, and is not needed by host libraries interfacing directly with the wire protocol.

The system EP11 interface consists of both host only APIs and APIs that interact directly with EP11 modules. Most functions interacting directly with the module have a PKCS#11 equivalent.

The host only API consist of queries (called host queries), management, configuration and helper functions.

The host library can interact with different platforms. The only platform supported officially is Linux for IBM z and only the interfaces for this platform are described here and in the wire format.

Please see the C file `ep11-app.c` in the documentation directory of the host library for an example for using the EP11 system interface.

10.1 The EP11 system interface

Most of the functions in the EP11 System interface have a PKCS#11 equivalent. See page 77 onwards for an overview of all PKCS#11 functions that are implemented by EP11. Page 79 and onwards lists and describes the functions defined in `ep11.h`. In general a `C_*()` PKCS#11 function directly maps to a corresponding `m_*()` EP11 function. Additional `m_*()` functions are provided in the system EP11 interface: Single calls, queries, the administrative interface and management and configuration functions.

The following sections describe the primitives regarding the EP11 system interface. Understanding them is crucial for using the interface.

10.1.1 Target tokens

Registering a module/domain combination, called a target, creates a target token which identifies this single target.

A target token may correspond to a specific single target or to a group of targets, then also referred to as target group.

Every EP11 system function that needs to interact with a module obtains destination information from the opaque target parameter. The host library maintains registered targets and target groups and allows it to add, update and remove them.

10.1.2 Routing requests

Different target tokens can be used to route requests to different modules and domains. Creating a target token does not imply that the route to the target is open and usable. However the host library can optionally probe a target before creating it.

Routing the request to one specific target of the target group is done by the underlying OS and therefore the rules how the requests are routed are not described here. Use cases are load balancing and having failsafe routes to modules if one route is not available.

How to registering modules, query already registered modules and create target groups is described in chapter 10.2.4 and 10.2.5.

10.1.3 EP11 blob specifications

Most EP11 functions have a prototype similar to their PKCS#11 equivilants, with some parameters replaced by blob specifications. Generally, parameters with PKCS#11 definitions (i.e., those with `CK_...` types) are passed verbatim from the PKCS#11 call. "Native" types, i.e., `unsigned char *` and `size_t` denote parameters that must be mapped from PKCS#11 entities to blobs.

Blobs are always represented as opaque start/length pairs, i.e., opaque `unsigned char` arrays, with a single byte count parameter (`size_t`). Each blob is initialized by a `generate`, `initialize`, or similar PKCS#11 "object creator" function. *Functions returning blobs* support a simplified version of PKCS#11 type queries: when passed a `NULL` blob pointer, the corresponding length parameter is set to the size of the returned blob. Blob creator functions never read, only write their byte count parameter (this is different from PKCS#11 buffer size handling).

Reported blob sizes may be slightly larger than necessary, and they are guaranteed to be sufficient with the supplied parameters. Blob sizes are in bytes.

Sequences of PKCS#11 calls generally track objects across *sessions* at the PKCS#11 level. Object blobs generally correspond to sessions, and the EP11-aware PKCS#11 layer must bind blobs to each session. Function descriptions provide descriptions on which PKCS#11 parameter maps to which blob, if ambiguous.

Blobs must be reused without modification, as they are opaque to the host. Blobs are internally typed (invisible to host code); module functions verify that they have been passed a blob of appropriate type, and reject incorrect ones. As discussed elsewhere in this document, blobs are integrity-protected through a module-specific MAC key, which is derived from the module transport wrapping key, so module groups' members accept each others MACed structures.

10.1.4 Template parsing

For functions which pass parameters through templates, template parsing is not comprehensive. *If a template contains multiple instances of the same attribute, the first one is accepted and used without checking for repetitions.* As the EP11-aware host library would need to manipulate template arrays, we assume that template inconsistencies are detected before calling EP11.

If attributes are missing from templates, defaults are substituted if defined by PKCS#11. Note that EP11 generally relies on very few template parameters, since usage restrictions and most other parameters are used by the host library, and ignored by EP11 itself.

10.1.5 Size queries

Most functions treat PKCS#11 size queries as a special case. PKCS#11 permits a conservative estimate of generated output ("number of bytes... may somewhat exceed the precise number of bytes needed"), therefore size queries may give estimates without performing actual cryptographic operations. (As an example, the size of PKCS-padded, encrypted data may be estimated to within one block's worth of bytes, even without decrypting it.)

For functions which support size queries, the following changes are made to transport:

- The host sets a boolean variable, indicating a size query.
- The host *may* send the length of input data, encoded as an integer, instead of actual data, if applicable. This depends on actual function, and obviously only applies if host data is used (i.e., does not apply to `DigestInit` calls, for example).

  Note that the number of fields is not changed, just the content of one or more data inputs. Whether this is actually done is function-dependent, but it is either always or never performed for any particular function (no ambiguity). This choice is transparent at the library interface, and may be implementation-dependent.

- Module code generally short-circuits processing when responding to a size query, skipping actual cryptographic operations.

  A few functions are exceptions, where returned data is limited and its overhead is not significant. These functions do not special-case size queries inside the module; the distinction is opaque to the host.

- The module returns the size of output, encoded as an integer, in the "returned data" field. As with encoding the length of incoming data, the number of fields does not change, only the interpretation of one of them.

  As described above, certain functions do not special-case size queries in the module. These return actual data even if responding to a size query, as the module is unaware of the nature of the host call.

- Host code parses the wire-encoded integer size, returns it to the caller, as described in the PKCS#11 specification [PKC04, 11.2].

Unlike blob size queries, PKCS#11 parameters (i.e., those with `CK_...` types) check the size of the output buffer (blobs never read, only write their bytecount). PKCS#11 parameters therefore may result in `CKR_BUFFER_TOO_SMALL`, while blob size queries can not.

Note that since size estimates do not perform cryptographic operations, they may report a size when the operation should fail. As an example, PKCS-padded encrypted data may be detected to be incorrect during decryption. In such a case, the actual operation returns a `CKR...` error, but the size query returns `CKR_OK` and a valid size.

10.1.6 Return values

We generally use return values according to PKCS#11 (regular `CKR_...` values), during functional calls. The following standard return values are used in similar, but not necessarily identical, cases to their PKCS#11 equivalents during functional calls:

- `CKR_FUNCTION_CANCELED` if an action is prohibited by the current control point setup. (No further details are provided.)
- `CKR_ATTRIBUTE_READ_ONLY` is returned if attributes within a modifiable, but only restrictable key blob may not be added (the same call *could* succeed if it tried to remove the same bits).
- `CKR_PIN_INCORRECT` if an invalid PIN blob is supplied to a call including one.
- `CKR_PIN_EXPIRED` if the—otherwise valid—PIN blob is not active within the module.
- `CKR_CANT_LOCK` if internal infrastructure, specifically locking, fails. These failures are not expected to be encountered in a properly working environment.

Note that these return values are defined for conditions which are not directly applicable to EP11. We use them under conditions related to their PKCS#11 originals.

Administration reserves its own return values, as most of those are outside PKCS#11 scope (see page 33).

In addition to standard return values, we define the following list as vendor extensions:

- `CKR_IBM_BLOB_ERROR` if a blob is malformed, such as it has an invalid MAC. This failure is higher priority than failures related to blob internals, such as if blob attributes conflict with a request (which follow their PKCS#11 rules).

  Note that we intentionally do not provide details about why the blob is considered invalid, to prevent attacks which differentiate between failures based on specific errors [Ble98].

- `CKR_IBM_WKID_MISMATCH` if the blob is rejected as it is controlled by a WK different from the active one. This error has a higher priority than `CKR_IBM_BLOB_ERROR`.

  Note that since the WKID is visible in blobs, and may be queried from the domain, separately reporting it does add a side channel (i.e., it only reports information already available to host-based attacker). A separate return code is available to help host code, since this condition has a well-defined recovery procedure: WK migration.

- `CKR_IBM_TRANSPORT_ERROR` if additional transport fields EP11 adds over PKCS#11—such as domains or function variants—are invalid, such as out of range.

  These failures roughly correspond to the PKCS#11 meaning of `CKR_ARGUMENTS_BAD`. They are separated to allow host code filter PKCS#11 errors—which may be user-relevant, and should be passed up to callers—and XCP-specific host ones.

- `CKR_IBM_INTERNAL_ERROR` indicates that the EP11 application has encountered an unexpected condition, and stopped processing. This includes unexpected failures of the underlying infrastructure, such as the CSP.

- CKR_IBM_BLOBKEY_CONFLICT is returned if a request returning wrapped state coincides with a WK change, and the WK at the time of rewrapping differs from that of blob arrival.

  In practical setups, hosts would procedurally preempt interleaving WK updates and functional requests. Under these conditions, hosts should not encounter this return value.
- CKR_IBM_MODE_CONFLICT is returned if a blob's own mode restrictions conflict with that of the system. It may also be reached if the operational mode changes while the request is being processed (in addition to checking during blob decryption).
- CKR_IBM_TARGET_INVALID indicated a invalid target token. This is not returned if a route to a target is not available.

For functions that do not return a CK_RV value, a XCP_OK or XCP_*E* value is returned as a regular integer. These are mostly the init/shutdown functions as well as the host module functions.

- XCP_OK function completed successfully.
- XCP_EINTERNAL internal state of the host library is invalid. This should never happen. The host library has encountered some unexpected conditions and stopped processing.
- XCP_EARG a function parameter is invalid
- XCP_ETARGET invalid target parameter
- XCP_EMEMORY memory allocation failed
- XCP_EMUTEX locking functions failed
- XCP_EINIT host library is not correctly initializes.
- XCP_EDEVICE an OS specific device driver error happend
- XCP_EGROUP target group is invalid. E.g. was updated or removed while a request was processed.
- XCP_ESIZE internal limits are exhausted.

For module management related errors, further error values are introduced:

- XCP_MOD_EOBSOLETE a specific feature or member field is no longer supported.
- XCP_MOD_EVERSION the used version for the module is too high or too low.
- XCP_MOD_EFLAGS some flags are not supported in general or in this specific environment.
- XCP_MOD_EMODULE the module number is invalid.
- XCP_MOD_EDOMAIN the domain index is invalid or out of range.
- XCP_MOD_EINIT the module is not initialized. E.g. the version field is not set.
- XCP_MOD_EPROBE the probe send to the module was not successful and the module could therefore not be added.

The host library logs system errors to syslog. Studying the syslog messages can help to understand the error correctly.

10.2 Beyond standard interfaces of the host library

Functions of the Host Library that have a direct PKCS#11 equivalent are only described in the API reference (chapter 10.2.5). Please see the PKCS#11 standard document for a full description.

Interfaces which are describes in this chapter are

- The administrative interface
- Single pass functions
- Module management
- Target group configuration

10.2.1 The administrative interface

The administrative interface consists of the single function function (`m_admin()`) in the `ep11.h` header file. All administrative services described in this document are used through this interface.

To facilitate usage of the0 interface, helper functions and wrapper functions are provides:

- administrative request and query builder functions
- helper functions for creating/parsing signer info and callback function for signing requests
- administrative response parsing (return value, reason code and payload)
- wrapper functions for querying the WK and setting a random WK
- helper functions for WK key part import
- helper functions for parsing and setting administrative attributes
- general `m_admin()` wrapper

See the `ep11adm.h` header file for more information.

10.2.2 Single functions

Single function process data in one pass and consolidate `m_Init` and `m_Encrypt`/`m_Decrypt`/`m_Sign`/`m_Verify` in one call. In the case the request size exceeds transport layer restrictions the host library does slice the payload in multiple requests.

10.2.3 Module and host queries

EP11 provides several types of queries that are not part of the PKCS#11standard to get (non-administrative) information about a module and domains. The host library provides the `m_get_xcp_info()` function to query a module or domain for information.

Types of queries that are supported are:

- API and build information (module and host)
- general information about the module (see chapeter 5.1.1 in the wire format)
- general information about domains
- information about extended capabilities
- which EC curves are supported
- audit log download

10.2.4 Module management

The host library maintains information about EP11 modules and domains. User can register a module with `m_add_module()`. Registering a module creates a target token which allows to send requests to the registered module. To register a target the module number and the domain index must be supplied by the user. Setting the same target will always return the same target token. `m_add_module()` can also be used to query information about already added modules and domains.

Modules can be deregistered with `m_rm_module()`. A valid target token can be used in a `m_*()` function to implicitly reregister a module, because module number and domain index are encoded into the target token.

More information about the two function can be found in the function reference. See also the information about the module and the target data structure/type in the wire format chapter 5.6.

10.2.5 Target group management

A target group is a set of targets that are maintained by the host library. Target groups are registered and configured through the same interface as modules. The underlying OS uses target groups to load balance EP11 requests and for failsafe scenarios if one target becomes unavailable.

On registration modules can be flagged *virtual*. Virtual modules and the targets described in the module are registered like before, but the target tokens identifying the targets will not be returned, but will be maintained as an entity, a target group, in the host library. Only a reference to this target group will be returned as a target token. For virtual modules more than one target can be parsed by `m_add_module()`. The host library limits the creation of target groups to a reasonable number. Target groups are a limited resource and they should be deregistered with `m_rm_module()` when they are no longer used.

Targets can be added or removed from the target group by calling `m_add_module()` or `m_rm_module()` with the target token of the target group.

The target token which references a target group can be used like a single token which identifies a single module/domain combination. Only administrative commands and queries are an exceptions. All functions working with administrative function will not work together with target groups, because in administrative functions it is important to route the request to an exact destination.

A special case is a target group with zero members. Using a target token which references a group with zero members is valid and can be useful, because the underlying OS will then use all available routes for its load balancing algorithm. However it has also drawbacks if different versions of EP11 modules are configured for the system. Some mechanism could only work on newer modules and fail on older modules. It must also be noted that all targets of a target group should be identically[2] configured.

[2]Should for example allow the same key sizes, algorithms and mechanisms and have the same Wrapping Key configured.

Appendix

Mechanisms, overview

The folowing listing contains all mechanisms currently supported by EP11

Mechanism	Encrypt	Decrypt	Digest	Sign	Verify	Generate	Wrap	Unwrap	Derive
CKM_AES_CBC	✓	✓					✓	✓	
CKM_AES_CBC_PAD	✓	✓					✓	✓	
CKM_AES_CMAC				✓	✓				
CKM_AES_ECB	✓	✓							
CKM_AES_KEY_GEN						✓			
CKM_DES2_KEY_GEN						✓			
CKM_DES3_CBC	✓	✓					✓	✓	
CKM_DES3_CBC_PAD	✓	✓					✓	✓	
CKM_DES3_CMAC				✓	✓				
CKM_DES3_ECB	✓	✓							
CKM_DES3_KEY_GEN						✓			
CKM_DH_PKCS_DERIVE									✓
CKM_DH_PKCS_KEY_PAIR_GEN						✓			
CKM_DH_PKCS_PARAMETER_GEN						✓			
CKM_DSA				✓	✓				
CKM_DSA_KEY_PAIR_GEN						✓			
CKM_DSA_PARAMETER_GEN						✓			
CKM_DSA_SHA1				✓	✓				
CKM_ECDH1_DERIVE									✓
CKM_ECDSA				✓	✓				
CKM_ECDSA_SHA1				✓	✓				
CKM_ECDSA_SHA224				✓	✓				
CKM_ECDSA_SHA256				✓	✓				
CKM_ECDSA_SHA384				✓	✓				
CKM_ECDSA_SHA512				✓	✓				
CKM_EC_KEY_PAIR_GEN						✓			
CKM_GENERIC_SECRET_KEY_GEN						✓			✓
CKM_IBM_ATTRIBUTEBOUND_WRAP							✓	✓	
CKM_IBM_CMAC				✓	✓				
CKM_IBM_CPACF_WRAP							✓		
CKM_IBM_DH_PKCS_DERIVE_RAW									✓
CKM_IBM_DILITHIUM				✓	✓	✓			
CKM_IBM_EAC									✓
CKM_IBM_ECDH1_DERIVE_RAW									✓
CKM_IBM_ECDSA_SHA224				✓	✓				
CKM_IBM_ECDSA_SHA256				✓	✓				
CKM_IBM_ECDSA_SHA384				✓	✓				
CKM_IBM_ECDSA_SHA512				✓	✓				
CKM_IBM_EC_MULTIPLY	✓								
CKM_IBM_EC_X25519									✓
CKM_IBM_EC_X448									✓
CKM_IBM_ED25519_SHA512				✓	✓				
CKM_IBM_ED448_SHA3				✓	✓				
CKM_IBM_RETAINKEY							✓		
CKM_IBM_SHA3_224			✓						
CKM_IBM_SHA3_224_HMAC				✓	✓				
CKM_IBM_SHA3_256			✓						
CKM_IBM_SHA3_256_HMAC				✓	✓				
CKM_IBM_SHA3_384			✓						
CKM_IBM_SHA3_384_HMAC				✓	✓				
CKM_IBM_SHA3_512			✓						
CKM_IBM_SHA3_512_HMAC				✓	✓				
CKM_IBM_SHA512_224			✓						
CKM_IBM_SHA512_224_HMAC				✓	✓				
CKM_IBM_SHA512_256			✓						
CKM_IBM_SHA512_256_HMAC				✓	✓				
CKM_PBE_SHA1_DES3_EDE_CBC						✓			
CKM_RSA_PKCS	✓	✓		✓	✓		✓	✓	
CKM_RSA_PKCS_KEY_PAIR_GEN						✓			
CKM_RSA_PKCS_OAEP	✓	✓					✓	✓	
CKM_RSA_PKCS_PSS				✓	✓				
CKM_RSA_X9_31				✓	✓				
CKM_RSA_X9_31_KEY_PAIR_GEN						✓			
CKM_SHA1_KEY_DERIVATION									✓
CKM_SHA1_RSA_PKCS				✓	✓				
CKM_SHA1_RSA_PKCS_PSS				✓	✓				
CKM_SHA1_RSA_X9_31				✓	✓				
CKM_SHA224			✓						
CKM_SHA224_HMAC				✓	✓				
CKM_SHA224_KEY_DERIVATION									✓
CKM_SHA224_RSA_PKCS				✓	✓				
CKM_SHA224_RSA_PKCS_PSS				✓	✓				
CKM_SHA256			✓						
CKM_SHA256_HMAC				✓	✓				
CKM_SHA256_KEY_DERIVATION									✓
CKM_SHA256_RSA_PKCS				✓	✓				
CKM_SHA256_RSA_PKCS_PSS				✓	✓				
CKM_SHA384			✓						
CKM_SHA384_HMAC				✓	✓				
CKM_SHA384_KEY_DERIVATION									✓
CKM_SHA384_RSA_PKCS				✓	✓				

CKM_SHA384_RSA_PKCS_PSS				✓	✓				
CKM_SHA512			✓						
CKM_SHA512_224			✓						
CKM_SHA512_224_HMAC				✓	✓				
CKM_SHA512_256			✓						
CKM_SHA512_256_HMAC				✓	✓				
CKM_SHA512_HMAC				✓	✓				
CKM_SHA512_KEY_DERIVATION									✓
CKM_SHA512_RSA_PKCS				✓	✓				
CKM_SHA512_RSA_PKCS_PSS				✓	✓				
CKM_SHA_1			✓						
CKM_SHA_1_HMAC				✓	✓				

Table 5: Mechanisms and function groups

List of mechanisms categorized by function group:

Encrypt: `CKM_AES_CBC` `CKM_AES_CBC_PAD` `CKM_AES_ECB` `CKM_DES3_CBC` `CKM_DES3_CBC_PAD` `CKM_DES3_ECB` `CKM_IBM_EC_MULTIPLY` `CKM_RSA_PKCS` `CKM_RSA_PKCS_OAEP`

Decrypt: `CKM_AES_CBC` `CKM_AES_CBC_PAD` `CKM_AES_ECB` `CKM_DES3_CBC` `CKM_DES3_CBC_PAD` `CKM_DES3_ECB` `CKM_RSA_PKCS` `CKM_RSA_PKCS_OAEP`

Digest: `CKM_IBM_SHA3_224` `CKM_IBM_SHA3_256` `CKM_IBM_SHA3_384` `CKM_IBM_SHA3_512` `CKM_IBM_SHA512_224` `CKM_IBM_SHA512_256` `CKM_SHA224` `CKM_SHA256` `CKM_SHA384` `CKM_SHA512` `CKM_SHA512_224` `CKM_SHA512_256` `CKM_SHA_1`

Sign: `CKM_AES_CMAC` `CKM_DES3_CMAC` `CKM_DSA` `CKM_DSA_SHA1` `CKM_ECDSA` `CKM_ECDSA_SHA1` `CKM_ECDSA_SHA224` `CKM_ECDSA_SHA256` `CKM_ECDSA_SHA384` `CKM_ECDSA_SHA512` `CKM_IBM_CMAC` `CKM_IBM_DILITHIUM` `CKM_IBM_ECDSA_SHA224` `CKM_IBM_ECDSA_SHA256` `CKM_IBM_ECDSA_SHA384` `CKM_IBM_ECDSA_SHA512` `CKM_IBM_ED25519_SHA512` `CKM_IBM_ED448_SHA3` `CKM_IBM_SHA3_224_HMAC` `CKM_IBM_SHA3_256_HMAC` `CKM_IBM_SHA3_384_HMAC` `CKM_IBM_SHA3_512_HMAC` `CKM_IBM_SHA512_224_HMAC` `CKM_IBM_SHA512_256_HMAC` `CKM_RSA_PKCS` `CKM_RSA_PKCS_PSS` `CKM_RSA_X9_31` `CKM_SHA1_RSA_PKCS` `CKM_SHA1_RSA_PKCS_PSS` `CKM_SHA1_RSA_X9_31` `CKM_SHA224_HMAC` `CKM_SHA224_RSA_PKCS` `CKM_SHA224_RSA_PKCS_PSS` `CKM_SHA256_HMAC` `CKM_SHA256_RSA_PKCS` `CKM_SHA256_RSA_PKCS_PSS` `CKM_SHA384_HMAC` `CKM_SHA384_RSA_PKCS` `CKM_SHA384_RSA_PKCS_PSS` `CKM_SHA512_224_HMAC` `CKM_SHA512_256_HMAC` `CKM_SHA512_HMAC` `CKM_SHA512_RSA_PKCS` `CKM_SHA512_RSA_PKCS_PSS` `CKM_SHA_1_HMAC`

Verify: `CKM_AES_CMAC` `CKM_DES3_CMAC` `CKM_DSA` `CKM_DSA_SHA1` `CKM_ECDSA` `CKM_ECDSA_SHA1` `CKM_ECDSA_SHA224` `CKM_ECDSA_SHA256` `CKM_ECDSA_SHA384` `CKM_ECDSA_SHA512` `CKM_IBM_CMAC` `CKM_IBM_DILITHIUM` `CKM_IBM_ECDSA_SHA224` `CKM_IBM_ECDSA_SHA256` `CKM_IBM_ECDSA_SHA384` `CKM_IBM_ECDSA_SHA512` `CKM_IBM_ED25519_SHA512` `CKM_IBM_ED448_SHA3` `CKM_IBM_SHA3_224_HMAC` `CKM_IBM_SHA3_256_HMAC` `CKM_IBM_SHA3_384_HMAC` `CKM_IBM_SHA3_512_HMAC` `CKM_IBM_SHA512_224_HMAC` `CKM_IBM_SHA512_256_HMAC` `CKM_RSA_PKCS` `CKM_RSA_PKCS_PSS` `CKM_RSA_X9_31` `CKM_SHA1_RSA_PKCS` `CKM_SHA1_RSA_PKCS_PSS` `CKM_SHA1_RSA_X9_31` `CKM_SHA224_HMAC` `CKM_SHA224_RSA_PKCS` `CKM_SHA224_RSA_PKCS_PSS` `CKM_SHA256_HMAC` `CKM_SHA256_RSA_PKCS` `CKM_SHA256_RSA_PKCS_PSS` `CKM_SHA384_HMAC` `CKM_SHA384_RSA_PKCS` `CKM_SHA384_RSA_PKCS_PSS` `CKM_SHA512_224_HMAC` `CKM_SHA512_256_HMAC` `CKM_SHA512_HMAC` `CKM_SHA512_RSA_PKCS` `CKM_SHA512_RSA_PKCS_PSS` `CKM_SHA_1_HMAC`

Generate: `CKM_AES_KEY_GEN` `CKM_DES2_KEY_GEN` `CKM_DES3_KEY_GEN` `CKM_DH_PKCS_KEY_PAIR_GEN` `CKM_DH_PKCS_PARAMETER_GEN` `CKM_DSA_KEY_PAIR_GEN` `CKM_DSA_PARAMETER_GEN` `CKM_EC_KEY_PAIR_GEN` `CKM_GENERIC_SECRET_KEY_GEN` `CKM_IBM_DILITHIUM` `CKM_PBE_SHA1_DES3_EDE_CBC` `CKM_RSA_PKCS_KEY_PAIR_GEN` `CKM_RSA_X9_31_KEY_PAIR_GEN`

Wrap: `CKM_AES_CBC` `CKM_AES_CBC_PAD` `CKM_DES3_CBC` `CKM_DES3_CBC_PAD` `CKM_IBM_ATTRIBUTEBOUND_WRAP` `CKM_IBM_CPACF_WRAP` `CKM_IBM_RETAINKEY` `CKM_RSA_PKCS` `CKM_RSA_PKCS_OAEP`

Unwrap: `CKM_AES_CBC` `CKM_AES_CBC_PAD` `CKM_DES3_CBC` `CKM_DES3_CBC_PAD` `CKM_IBM_ATTRIBUTEBOUND_WRAP` `CKM_RSA_PKCS` `CKM_RSA_PKCS_OAEP`

Derive: `CKM_DH_PKCS_DERIVE` `CKM_ECDH1_DERIVE` `CKM_GENERIC_SECRET_KEY_GEN` `CKM_IBM_DH_PKCS_DERIVE_RAW` `CKM_IBM_EAC` `CKM_IBM_ECDH1_DERIVE_RAW` `CKM_IBM_EC_X25519` `CKM_IBM_EC_X448` `CKM_SHA1_KEY_DERIVATION` `CKM_SHA224_KEY_DERIVATION` `CKM_SHA256_KEY_DERIVATION` `CKM_SHA384_KEY_DERIVATION` `CKM_SHA512_KEY_DERIVATION`

PKCS#11 wire functions

This section lists all PKCS#11 wire functions for reference. For more information please see the wire format

Function (index)		*Inputs*	*Outputs*	*Int. outputs*	*Parameters*
`Login`	1	2	1	0b0	PIN, (nonce)
					PIN blob
`Logout`	2	1	0	-	PIN blob
					only return value
`SeedRandom`	3	1	0	-	new seed
					only return value
`GenerateRandom`	4	1	1	0b0	bytecount
					rnbytes
`DigestInit`	5	2	1	0b0	fnvariant, mech (digest)
					state (hash)
`DigestUpdate`	6	2	1	0b0	state (hash), data
					state (hash)
`DigestKey`	7	2	1	0b0	state (hash), keyblob
					state (hash)
`DigestFinal`	8	2	1	0b0	size query? (boolean), state (hash)
					digest
`Digest`	9	3	1	0b0	size query? (boolean), state (hash), data
					digest
`EncryptInit`	11	3	1	0b0	var, mech (crypt), keyblob
					state (crypt)
`DecryptInit`	12	3	1	0b0	var, mech (crypt), keyblob
					state (crypt)
`EncryptUpdate`	13	3	2	0b10	size query? (boolean), state (crypt), data
					state, (output)
`DecryptUpdate`	14	3	2	0b10	size query? (boolean), state (crypt), data
					state, (output)
`EncryptFinal`	15	2	1	0b0	size query? (boolean), state (crypt)
					ciphertext
`DecryptFinal`	16	2	1	0b0	size query? (boolean), state (crypt)
					plaintext
`Encrypt`	17	3	1	0b0	size query? (boolean), state (crypt), plaintext
					ciphertext
`Decrypt`	18	3	1	0b0	size query? (boolean), state (crypt), ciphertext
					plaintext
`GenerateKey`	21	5	2	0b00	fn variant, key bytes, mech (symm gen), attributes, (pinblob)
					keyblob, checksum
`GenerateKeyPair`	22	4	2	0b00	mech (PK gen), public attributes, private attributes, (pinblob)
					keyblob, SPKI
`SignInit`	23	3	1	0b0	variant, mech (sign), keyblob
					state (sign)
`SignUpdate`	24	2	1	0b0	data, state (sign)
					state (sign)
`SignFinal`	25	2	1	0b0	size query? (boolean), state (sign)
					signature
`Sign`	26	3	1	0b0	size query? (boolean), state (sign), data
					signature
`VerifyInit`	27	3	1	0b0	variant, mech (sign), keyblob
					state (verify)
`VerifyUpdate`	28	2	1	0b0	state (verify), data

				state (verify)
`VerifyFinal` 29	2	0	-	state (verify), signature
				only return value
`Verify` 30	3	0	-	state (verify), signature, data
				only return value
`WrapKey` 33	5	1	0b0	fn variant, mech (wrap), key, wrapkey, (MAC key)
				wrapped key
`UnwrapKey` 34	6	2	0b00	new key attributes, mechanism (wrap), KEK blob, (MAC key), (pinblob), wrapped key
				keyblob, (checksum)
`DeriveKey` 35	5	2	0b00	derivation alg, new key attrs, base key, (pinblob), data(blob)
				keyblob, (checksum)
`GetMechanismList` 36	0	1	0b0	
				mechanisms
`GetMechanismInfo` 37	1	1	0b0	mechanism
				mech info
`GetAttributeValue` 39	2	1	0b0	keyblob, attrlist
				attribute values
`SetAttributeValue` 40	2	1	0b0	keyblob, attribute values
				updated keyblob

Table 6: PKCS#11 function equivalents

Auxiliary wire functions

This section lists all auxiliary wire functions for reference. For more information please see the wire format

Function (index)	*Inputs*	*Outputs*	*Int. outputs*	*Parameters*
`DigestSingle` 10	3	1	0b0	size query? (boolean), mech (hash), data
				digest
`EncryptSingle` 19	4	1	0b0	variant, mech (crypt), keyblob, plaintext
				ciphertext
`DecryptSingle` 20	4	1	0b0	variant, mech (crypt), keyblob, ciphertext
				plaintext
`SignSingle` 31	4	1	0b0	size query? (boolean), mech (sign), keyblob, data
				signature
`VerifySingle` 32	4	0	-	mech (sign), keyblob/SPKI, data, signature
				only return value
`get_xcp_info` 38	2	1	0b0	query, subquery
				module/domain info
`admin` 41	2	2	0b00	payload, signature/s
				response1, response2

Table 7: Auxiliary functions

Function overview

The function listing is grouped by functionality. It does not contain non-standard extensions, such as `_...Single` calls, but references them where appropriate.

Functions without notes are implemented in a PKCS#11 compliant manner, without further discussion. Functions with names in bold are implemented and callable; the remaining functions are not applicable to EP11 and they do not exist in the library.

Function	Notes
C_EncryptInit	
C_Encrypt	see also **_EncryptSingle**
C_EncryptUpdate	
C_EncryptFinal	
C_DecryptInit	
C_Decrypt	see also **_DecryptSingle**
C_DecryptUpdate	
C_DecryptFinal	
C_DigestInit	creates wrapped state objects
C_Digest	see also **_DigestSingle**
C_DigestUpdate	
C_DigestKey	not implemented
C_DigestFinal	
C_SignInit	
C_Sign	see also **_SignSingle**
C_SignUpdate	
C_SignFinal	
C_VerifyInit	
C_Verify	see also **_VerifySingle**
C_VerifyUpdate	
C_VerifyFinal	
C_GenerateKey	
C_GenerateKeyPair	
C_WrapKey	enforces relative key size restrictions
C_UnwrapKey	
C_DeriveKey	
C_GenerateRandom	no external seeding unless specified explicitly
C_SeedRandom	not supported unless specified by build setting
C_GetMechanismList	
C_GetMechanismInfo	
C_WaitForSlotEvent	N/A; “slot” is not removable
C_SignRecoverInit	not implemented
C_SignRecover	not implemented
C_VerifyRecoverInit	not implemented
C_VerifyRecover	not implemented
C_DigestEncryptUpdate	not implemented
C_DecryptDigestUpdate	not implemented
C_SignEncryptUpdate	not implemented
C_DecryptVerifyUpdate	not implemented
C_GetInfo	not supported; features not relevant in traditional PKCS#11 terms
C_GetFunctionList	not supported; functions interfaces are not standard PKCS#11
C_GetSlotList	not implemented
C_GetSlotInfo	not implemented
C_GetTokenInfo	not implemented, see get_xcp_info for partial support
C_Initialize	
C_Finalize	
C_GetOperationState	N/A, operation state resides on host
C_SetOperationState	N/A, operation state resides on host
C_GetFunctionStatus	N/A, function state resides on host

C_CancelFunction	N/A, function state resides on host
C_GetAttributeValue	restricted to backend-visible attributes
C_SetAttributeValue	restricted to backend-visible attributes
C_FindObjectsInit	N/A, tokens/sessions reside on host
C_FindObjects	N/A, tokens/sessions reside on host
C_FindObjectsFinal	N/A, tokens/sessions reside on host
C_InitToken	not implemented
C_InitPIN	not implemented
C_SetPIN	not implemented
C_OpenSession	N/A, sessions are managed by host
C_CloseSession	N/A, sessions are managed by host
C_CloseAllSessions	N/A, sessions are managed by host
C_GetSessionInfo	N/A, sessions are managed by host
C_Login	
C_Logout	
C_CreateObject	not implemented; can be approximated by import
C_CopyObject	N/A, objects are handled on host
C_DestroyObject	N/A, objects are handled on host
C_GetObjectSize	N/A, objects are handled on host

Table 8: Function groups

Function reference

Highlighted functions have no PKCS#11 equivalent.

Equivalents of standard PKCS#11 functions are not described in detail if their interface or implementation is obvious, or corresponds to PKCS#11 in a straightforward manner. EP11 follows the PKCS#11 version 2.20 at the moment.

```
[*]   m_add_module
[*]   m_admin
      m_Decrypt
      m_DecryptFinal
      m_DecryptInit

[*]   m_DecryptSingle
      m_DecryptUpdate
      m_DeriveKey
      m_Digest
      m_DigestFinal

      m_DigestInit
      m_DigestKey
[*]   m_DigestSingle
      m_DigestUpdate
      m_Encrypt

      m_EncryptFinal
      m_EncryptInit
[*]   m_EncryptSingle
      m_EncryptUpdate
      m_GenerateKey

      m_GenerateKeyPair
      m_GenerateRandom
[*]   m_get_ep11_info
[*]   m_get_xcp_info
      m_GetAttributeValue

      m_GetMechanismInfo
      m_GetMechanismList
[*]   m_init
      m_Login
      m_Logout

[*]   m_rm_module
      m_SeedRandom
      m_SetAttributeValue
[*]   m_shutdown
      m_Sign

      m_SignFinal
      m_SignInit
[*]   m_SignSingle
      m_SignUpdate
      m_UnwrapKey

      m_Verify
      m_VerifyFinal
      m_VerifyInit
[*]   m_VerifySingle
      m_VerifyUpdate

[*]   m_wire
```

m_WrapKey

m_add_module

Allows to modify and query a module. The module data structure needs to contain the target, the module number/domain index combination. Returns a target token identifying the newly registered target.

Setting the same target twice does yield the same target token.

Setting another target in the module data structure after the module was registered returns a target token identifying the new target. Already added targets are ignored.

If the module is flagged with XCP_MFL_VIRTUAL a target group is created or updated together with registering or updating the module. A virtual module can have more than one target enabled and as a special case can have zero targets enabled.

Modules are checked for consistency and small errors are corrected. Please see chapter 5.6.3 in the Wire Format for a summary of errors that are correctable. Use XCP_MFL_STRICT to make m_add_module return errors instead of correcting them.

The module parameter must always be valid. To query the registered modules set target to NULL and set the module_nr of the module to the number you want to query. The version needs also to be set and the backend flag for the corresponding platform needs to be set (see chapter 5.6.3.1 of the wire format) XCP_EINIT is returned if the module is not registered. Otherwise the module is filled with additional information.

Please check the availability of this function by querying CK_IBM_XCPHQ_TGT_MODE.

```
int m_add_module(XCP_Module_t module, target_t *target) ;
```

m_admin

Non-PKCS11 function. Combine administrative request block with [optional] signatures, submit to backend. Common interface to all administrative services, which are internally dispatched to queries/commands based on request block contents.

The ASN.1-encoded request block must be supplied within `(cmd, clen)`; see the wire spec for request-block construction rules.

Signature[s], if present, MUST all correspond to the same request block; ASN.1-encoded signature `signerInfo`s MUST be simply concatenated within `(sigs, slen)`, without further formatting or padding. `NULL` as `sigs` implies no signatures on request, such as queries, or un-signed commands. Since signature structures specify their signers, algorithms etc. within each `signerInfo`, and administrative services infer their authorized signers based on the request block, no further signature-related information is needed.

Returns full response block in `(response1, r1len)`. Module signature, if present, is returned as an ASN.1-encoded `signerInfo` struct in `(response2, r2len)`. The signature covers the preceding response block. In-band administrative signatures, such as those within exported module state, are within the response block, outside `response2`.

In case of administrative failures, this function MAY return `CKR_OK` and the proper [administrative] error [non-`OK`] is returned within the response block. This special case has been added to preserve the top-level goal of returning only an error value without data, in case of failures.

Administrative errors are considerably more specific than PKCS11 ones, which is why they MAY be returned through indirection. Generally, wire-level failures are reported regularly, but errors under specific administrative conditions are reported through indirection. See response-block rules in the wire spec for details.

Note that the host function does not validate any signatures, number of signature structures etc., it simply serializes inputs to pass to the module.

```
CK_RV m_admin (unsigned char *response1, size_t *r1len,
               unsigned char *response2, size_t *r2len,
         const unsigned char *cmd,       size_t clen,
         const unsigned char *sigs,      size_t slen,
                    target_t target) ;
```

m_Decrypt

Implementation of PKCS#11 `C_Decrypt`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `DecryptInit`.

```
CK_RV m_Decrypt (const unsigned char *state,       size_t slen,
                        CK_BYTE_PTR cipher,     CK_ULONG clen,
                        CK_BYTE_PTR plain,  CK_ULONG_PTR plen,
                           target_t target) ;
```

m_DecryptFinal

Implementation of PKCS#11 `C_DecryptFinal`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `DecryptInit`, `DecryptUpdate`.

```
CK_RV m_DecryptFinal  (const unsigned char *state,       size_t slen,
                             CK_BYTE_PTR output, CK_ULONG_PTR len,
                                target_t target) ;
```

m_DecryptInit

Implementation of PKCS#11 C_DecryptInit.

```
CK_RV m_DecryptInit (unsigned char *state, size_t *slen,
                  CK_MECHANISM_PTR pmech,
               const unsigned char *key,   size_t klen,
                          target_t target) ;
```

m_DecryptSingle

Non-standard variant of Decrypt. Processes data in one pass, with one call. Does not return any state to host, only decrypted data.

This is the preferred method of encrypting data in one pass for XCP-aware applications. Functionally it is equivalent to DecryptInit followed immediately by Decrypt, but it saves roundtrips and wrapping/unwrapping.

If the backend supports resident keys, the key may be also a resident-key handle.

See also: Decrypt, DecryptInit, EncryptSingle.

The key blob was output from: GenerateKey, UnwrapKey.

```
CK_RV m_DecryptSingle (const unsigned char *key,         size_t klen,
                          CK_MECHANISM_PTR pmech,
                               CK_BYTE_PTR cipher,     CK_ULONG clen,
                               CK_BYTE_PTR plain,  CK_ULONG_PTR plen,
                                  target_t target) ;
```

m_DecryptUpdate

Implementation of PKCS#11 C_DecryptUpdate.

The state,slen blob must be mapped from the PKCS11 hSession parameter.

The state blob was output from: DecryptInit.

```
CK_RV m_DecryptUpdate (unsigned char *state,       size_t slen,
                        CK_BYTE_PTR cipher,     CK_ULONG clen,
                        CK_BYTE_PTR plain,  CK_ULONG_PTR plen,
                           target_t target) ;
```

m_DeriveKey

Implementation of PKCS#11 C_DeriveKey.

The basekey,bklen blob must be mapped from the PKCS11 hBaseKey parameter.

PKCS#11 hSession is not mapped to any EP11 parameter. (The call is not directly associated with any session.)

PKCS#11 phKey is not mapped to any EP11 parameter. (Host library must bind returned key to handle.)

```
CK_RV m_DeriveKey ( CK_MECHANISM_PTR pderivemech,
                    CK_ATTRIBUTE_PTR ptempl, CK_ULONG templcount,
                  const unsigned char *basekey, size_t bklen,
                  const unsigned char *data,    size_t dlen,
                  const unsigned char *pin,     size_t pinlen,
                        unsigned char *newkey,  size_t *nklen,
                        unsigned char *csum,    size_t *cslen,
                        target_t target) ;
```

m_Digest

Implementation of PKCS#11 `C_Digest`.

Note that if a digest object has had exactly 0 (zero) bytes appended to it after creation, in any combination of zero byte transfers, it may still perform a one-pass Digest, even if it should be rejected by a strict implementation. This is a feature.

Does not update `(state, slen)`.

Implementations *may* perform `DigestUpdate`, `DigestFinal`, or `Digest` calls on cleartext digest objects in host code, bypassing HSM backends altogether. This choice may or may not be visible to host code, and it does not impact the security of the operation (as clear objects can not digest sensitive data).

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `DigestInit`.

```
CK_RV m_Digest (const unsigned char *state,       size_t slen,
                          CK_BYTE_PTR data,       CK_ULONG len,
                          CK_BYTE_PTR digest, CK_ULONG_PTR dglen,
                             target_t target) ;
```

m_DigestFinal

Implementation of PKCS#11 C_DigestFinal.

DigestFinal is polymorphic, accepting both wrapped or clear digest objects.

Does not update (state, slen).

The state,slen blob must be mapped from the PKCS11 hSession parameter.

The state blob was output from: DigestInit, DigestUpdate, DigestKey.

note: does not have data, size query changes no fields' meaning

```
CK_RV m_DigestFinal(const unsigned char *state,       size_t slen,
                            CK_BYTE_PTR digest, CK_ULONG_PTR dlen,
                               target_t target) ;
```

m_DigestInit

Implementation of PKCS#11 `C_DigestInit`.

Create wrapped digest state.

Note: size queries are supported, but the wrapped state is always returned by the backend, unlike most size queries (which return an output size, instead of actual output). `Digest` states are sufficiently small that they do not introduce noticeable transport overhead, except obviously object wrapping.

During size queries, the host just discards the returned state, and reports blob size (in `len`). When returning blob, *len is checked against returned size.

The `state,len` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must tie blob to session.)

```
CK_RV m_DigestInit(unsigned char *state, size_t *len,
          const CK_MECHANISM_PTR pmech,
                        target_t target) ;
```

m_DigestKey

Implementation of PKCS#11 C_DigestKey.

Note that, by construction, none of the DigestKey inputs are native PKCS#11 variables. Both are blobs, one containing wrapped state, the other a key object. No final result is returned, only state is updated during the call.

DigestKey is the only non-polymorphic DigestNNN call. It rejects clear digest objects, since those could reveal key bytes when passed back to the host in the clear.

The state,slen blob must be mapped from the PKCS11 hSession parameter.

The state blob was output from: DigestInit, DigestUpdate, DigestKey.

```
CK_RV m_DigestKey    (unsigned char *state, size_t slen,
               const unsigned char *key,   size_t klen,
                     target_t target) ;
```

m_DigestSingle

Nonstandard extension, combination of `DigestInit` and `Digest`. Digests data in one pass, with one call, without constructing an intermediate digest state, and unnecessary roundtrips.

This is the preferred method of digesting cleartext for XCP-aware applications. Functionally, `DigestSingle` is equivalent to `DigestInit` followed immediately by `Digest`.

If a key needs to be digested, one *must* use `DigestInit` and `DigestKey`, since this function does not handle key blobs.

Does not return any state to host, only digest result. There are no non-PKCS#11 parameters, since everything is used directly from the PKCS#11 call.

```
CK_RV m_DigestSingle(CK_MECHANISM_PTR pmech,
                         CK_BYTE_PTR data,   CK_ULONG len,
                         CK_BYTE_PTR digest, CK_ULONG_PTR dlen,
                            target_t target) ;
```

m_DigestUpdate

Implementation of PKCS#11 C_DigestUpdate.

DigestUpdate is polymorphic, accepting both wrapped or clear digest objects, updating state in the same format.

The state,slen blob must be mapped from the PKCS11 hSession parameter. (Host library must map session to stored state.)

The state blob was output from: DigestInit, DigestUpdate, DigestKey.

See also: DigestInit

```
CK_RV m_DigestUpdate(unsigned char *state,  size_t slen,
                       CK_BYTE_PTR data,  CK_ULONG dlen,
                          target_t target) ;
```

m_Encrypt

Implementation of PKCS#11 `C_Encrypt`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `EncryptInit`.

```
CK_RV m_Encrypt (const unsigned char *state,       size_t slen,
                       CK_BYTE_PTR plain,      CK_ULONG plen,
                       CK_BYTE_PTR cipher, CK_ULONG_PTR clen,
                           target_t target) ;
```

m_EncryptFinal

Implementation of PKCS#11 `C_EncryptFinal`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `EncryptInit`, `EncryptUpdate`.

```
CK_RV m_EncryptFinal  (const unsigned char *state,       size_t slen,
                             CK_BYTE_PTR output, CK_ULONG_PTR len,
                                target_t target) ;
```

m_EncryptInit

Implementation of PKCS#11 C_EncryptInit.

The (key, klen) blob may be a public-key object, or a secret-key blob. Key type must be consistent with pmech.

For public-key mechanisms, (key, klen) must contain an SPKI. This SPKI *must be MACed*, as returned by GenerateKeyPair or alternatively UnwrapKey. The Encrypt state is created without session restrictions.

For secret-key mechanisms, the Encrypt state inherits object session restrictions from <prm>key,klen<prm>.

The state,slen blob must be mapped from the PKCS11 hSession parameter.

(key, klen) must be a key blob.

```
CK_RV m_EncryptInit (unsigned char *state, size_t *slen,
                  CK_MECHANISM_PTR pmech,
               const unsigned char *key,   size_t klen,
                          target_t target) ;
```

m_EncryptSingle

Non-standard variant of Encrypt. Processes data in one pass, with one call. Does not return any state to host, only encrypted data.

This is the preferred method of encrypting data in one pass for XCP-aware applications. Functionally it is equivalent to EncryptInit followed immediately by Encrypt, but it saves roundtrips and wrapping/unwrapping.

If the backend supports resident keys, the key may be also a resident-key handle.

See also: Encrypt, EncryptInit, DecryptSingle.

The key blob was output from: GenerateKey, UnwrapKey.

```
CK_RV m_EncryptSingle (const unsigned char *key,         size_t klen,
                          CK_MECHANISM_PTR pmech,
                               CK_BYTE_PTR plain,      CK_ULONG plen,
                               CK_BYTE_PTR cipher, CK_ULONG_PTR clen,
                                  target_t target) ;
```

m_EncryptUpdate

Implementation of PKCS#11 `C_EncryptUpdate`.

The `state,slen` blob must be mapped from the PKCS11 `hSession` parameter.

The `state` blob was output from: `EncryptInit`.

```
CK_RV m_EncryptUpdate (unsigned char *state,       size_t slen,
                         CK_BYTE_PTR plain,      CK_ULONG plen,
                         CK_BYTE_PTR cipher, CK_ULONG_PTR clen,
                       target_t target) ;
```

m_GenerateKey

Implementation of PKCS#11 C_GenerateKey.

TDES keys are generated with proper parity. This is not observable by the host, but it is needed for proper interoperability: other PKCS#11 implementations *should* reject DES keys with parity problems.

If tying an object to a session, (pin, plen) must have been returned by Login to that session. Leaving pin NULL creates a public object, one not bound to a login session.

(key, klen) will return the key blob. (csum, clen) will contain the key's checksum, i.e., the most significant bytes of an all-zero block encrypted by the key. NULL clen is possible, for example for symmetric-key mechanisms without CKA_CHECK_VALUE parameters (such as RC4).

ptempl is used only if the key length (i.e., the CKA_VALUE_LEN attribute) is needed by the mechanism. If the mechanism implicitly specifies key size, ptempl is not checked for size.

DSA and DH parameter generation ignores (csum, clen), generating only parameter structures.

DSA,DH parameters (CKM_DSA_PARAMETER_GEN etc.): pass modulus bitcount in CKA_PRIME_BITS of attributes. Writes P,Q,G structure as cleartext output (i.e., not a blob).

The pin blob was output from: Login.

PKCS#11 phKey is not mapped to any EP11 parameter. (Host library must bind wrapped key to handle.)

```
CK_RV m_GenerateKey (CK_MECHANISM_PTR pmech,
                     CK_ATTRIBUTE_PTR ptempl, CK_ULONG templcount,
                  const unsigned char *pin,     size_t pinlen,
                        unsigned char *key,     size_t *klen,
                        unsigned char *csum,    size_t *clen,
                             target_t target) ;
```

m_GenerateKeyPair

Implementation of PKCS#11 C_GenerateKeyPair.

Keypair parameters are retrieved from pmech, ppublic, and pprivate parameters. For RSA keys, ppublic specifies the modulus size.

In FIPS mode, only RSA moduluses of 1024+256n bits are supported (integer n). Non-FIPS mode can generate keys of any even number of bits between the limits in the mechanism parameter list.

Public key is formatted as a standard SPKI (subject publickey info), readable by most libraries. It is integrity-protected by a transport-key specific MAC, which is not part of the SPKI itself. DSA parameter generation returns a non-SPKI structure in the public key field.

If tying an object to a session, (pin, plen) must have been returned by Login to that session. Leaving pin NULL creates a public object, one which will survive its login session.

Returns wrapped private key to (key, klen), public key as a MACed ASN.1/DER structure in (pubkey, pklen).

Supported parameter combinations with special notes (beyond those documented by PKCS11) are the following:

RSA keys reject public exponents below 17 (0x11). Control points may further restrict the accepted minimum. The Fermat4 exponent, 0x10001, is controlled by a specific control point, matching public-exponent restrictions of FIPS 186-3 (section B.3.1).

EC keys (CKM_EC_KEY_PAIR_GEN): curve parameters may be specified as OIDs or symbolic names (our namedCurve variant). Supported symbolic names are "P-nnn" for FP NIST curves (nnn is a supported prime bitcount, 192 to 521), "BP-nnnR" for regular BP curves, and "BP-nnnT" for twisted BP curves of bitcount nnn (160 to 512). (Names must be supplied as ASCII strings, without zero-termination.)

DSA keys (CKM_DSA_KEY_PAIR_GEN): pass P,Q,G structure as the CKA_IBM_STRUCT_PARAMS attribute of public attributes. Note that individual P,Q,G parameters may not be passed through regular PKCS#11 parameters, they must be combined to a single structure.

DH keys (CKM_DH_PKCS_KEY_PAIR_GEN): pass P,G structure as the CKA_IBM_STRUCT_PARAMS attribute of public attributes. Note that individual P,G parameters may not be passed through regular PKCS#11 parameters, they must be combined to a single structure. When selecting a private-key (X) bitcount, use the XCP_U32_VALUE_BITS attribute. If not present, or an explicit 0 is supplied, bitcount is selected based on P bitcount.

Use of session (Login) state replaces standard use of sessions, the mapping is outside library scope.

The pin blob was output from: Login.

PKCS#11 hSession is not mapped to any EP11 parameter. (The call is not directly associated with any session.)

PKCS#11 phPublicKey is not mapped to any EP11 parameter. (Host library must associate pubkey (SPKI) with handle.)

PKCS#11 phPrivateKey is not mapped to any EP11 parameter. (Host library must associate private key with handle.)

```
CK_RV m_GenerateKeyPair (CK_MECHANISM_PTR pmech,
                         CK_ATTRIBUTE_PTR ppublic,  CK_ULONG pubattrs,
                         CK_ATTRIBUTE_PTR pprivate, CK_ULONG prvattrs,
                      const unsigned char *pin,       size_t pinlen,
                            unsigned char *key,       size_t *klen,
                            unsigned char *pubkey,    size_t *pklen,
                                 target_t target) ;
```

m_GenerateRandom

Implementation of PKCS#11 `C_GenerateRandom`.

`GenerateRandom` is equivalent to the original PKCS#11 function. Internally, hardware-seeded entropy is passed through a FIPS-compliant DRNG (ANSI X9.31/ISO 18031, depending on Clic version).

The host library *could* generate random numbers without dispatching to the backend, if suitable functionality would be available on the host. This is not done in the current implementation.

This function does not support a size query.

```
CK_RV m_GenerateRandom (CK_BYTE_PTR rnd, CK_ULONG len, target_t target) ;
```

m_get_ep11_info

see m_get_xcp_info

```
CK_RV m_get_ep11_info(CK_VOID_PTR pinfo, CK_ULONG_PTR infbytes,
                      unsigned int query,
                      unsigned int subquery,
                      target_t target) ;
```

m_get_xcp_info

Non-PKCS11: polymorphic query function, selecting response based on `query`, and optionally, the further sub-parameter `subquery`. See `CK_IBM_XCPQUERY_t` for the list of available queries.

Wire-decoded outputs are deposited into `pinfo` if non-`NULL`, casting into a struct of the appropriate type where applicable. In such cases, mirroring PKCS11 parameter/size usage, provide at least `sizeof(struct ...)` in `infbytes`. After successful calls, `infbytes` is set to response-written bytecount.

Currently, the following structure types MAY be returned, and write a struct into `*pinfo` if non-NULL: `CK_IBM_XCPAPI_INFO` `CK_IBM_XCP_INFO` `CK_IBM_DOMAIN_INFO` `XCP_EPX_INFO` The returned structure is selected based on the query type, filling the entire structure.

In addition to queries returning types, the following [sub-]queries return lists of wire-encoded integers: `CK_IBM_XCPQ_SELFTEST` `CK_IBM_XCPQ_EXT_CAPS` `CK_IBM_XCPQ_EXT_CAPLIST` `CK_IBM_XCPQ_EPX` These variants return [potentially lists of] 32-bit unsigned integers; see the wire spec for integer-encoding rules.

Structure outputs are fixed size, except `CK_IBM_XCPQ_DOMAINS`, which returns a list. We check response size `infbytes`, which MAY be larger than needed by structs. Regular PKCS11 report-size rules apply.

```
CK_RV m_get_xcp_info(CK_VOID_PTR pinfo, CK_ULONG_PTR infbytes,
                    unsigned int query,
                    unsigned int subquery,
                        target_t target) ;
```

m_GetAttributeValue

Implementation of PKCS#11 `C_GetAttributeValue`.

Does not represent/need sessions (part of blob), therefore does not use the `hSession` parameter.

Since currently, we can do shortcuts (such as enumerate actual values instead of being more generic), decoding is straightforward.

```
CK_RV m_GetAttributeValue (const unsigned char *obj,        size_t olen,
                              CK_ATTRIBUTE_PTR pTemplate, CK_ULONG ulCount,
                                      target_t target) ;
```

m_GetMechanismInfo

Implementation of PKCS#11 `C_GetMechanismInfo`.

```
CK_RV m_GetMechanismInfo (CK_SLOT_ID slot,
                  CK_MECHANISM_TYPE mech,
              CK_MECHANISM_INFO_PTR pmechinfo,
                           target_t target) ;
```

m_GetMechanismList

Implementation of PKCS#11 `C_GetMechanismList`.

```
CK_RV m_GetMechanismList (CK_SLOT_ID slot,
               CK_MECHANISM_TYPE_PTR mechs,
                        CK_ULONG_PTR count,
                            target_t target) ;
```

m_init

Called before any functional calls, `m_init` performs startup sanity checks and initializes the transport channel.

The EP11 host library uses the AP device driver to send and receive messages from EP11 Crypto Express Cards (CEX). Please ensure that the driver is loaded when calling this function.

This function is not thread safe.

`XCP_DEV_ERROR` is returned if there is a problem with the the AP device driver. `XCP_INIT_ERROR` is returned if the library is already initialized.

```
int m_init(void) ;
```

m_Login

Implementation of PKCS#11 C_Login.

Turn caller-supplied PIN, and optionally a nonce, into a session identifier, a cryptographic signed representation of this combination of inputs. The returned session identifier may be used to bind generated, derived, or imported keys to that session, preventing use of these keys when the session is subsequently removed.

When session-bound keys are created, only a part of the session is inserted into the host-visible form. This subset is sufficient to identify sessions, but insufficient to log that session out.

Depending on represented host entities, a PIN may have a meaning of applications, partitions, jobs, or other similar constructs. Since the backend lacks understanding of these entities, it only treats PIN+nonce pairs as *entities* without further differentiation.

The nonce or some representative derivative appears in the generated session identifier, when such binding needs to be represented to the original caller.

In case the provided PIN blob buffer is too small the session will be directly logged out again in order to prevent exhaustion of session related card resources.

See also: Logout

```
CK_RV m_Login ( CK_UTF8CHAR_PTR pin,      CK_ULONG pinlen,
            const unsigned char *nonce,     size_t nlen,
                  unsigned char *pinblob,   size_t *pinbloblen,
                  target_t target) ;
```

m_Logout

Implementation of PKCS#11 C_Logout.

Remove the session derived by Login, if present within the targeted module.

This call requires access to the full session identifier. As an alternative, see the administrative command which may remove any session. The administrative command is not available to regular users or PKCS11 officers.

The pin blob was output from: Login.

```
CK_RV m_Logout(const unsigned char *pin, size_t len, target_t target) ;
```

m_rm_module

The `m_rm_module` function deregisters `modules`, removes targets from target groups or removes them completely.

If the target token is not initialized (is equal `XCP_TGT_INIT`) the targets defined in `module` are deregistered. No resource deallocation is done. Deregistration is only a simple bookkeeping function. Use `m_shutdown` to remove all modules associated channel resources.

To remove a target group the `target` parameter needs to refer to a target group. If the `module` parameter is NULL the complete group is removed. Otherwise only the targets from the module are removed from the group.

Please note that single targets can simply be discarded without host library interaction. Please also note that a valid target token can still be used after deregistration when `XCP_MFL_ALW_TGT_ADD` is set for the module.

Deregistration does not influence target groups and vice versa.

Please check the availability of this function by querying `CK_IBM_XCPHQ_TGT_MODE`.

```
int m_rm_module(const XCP_Module_t module, target_t target) ;
```

m_SeedRandom

Implementation of PKCS#11 `C_SeedRandom`.

`SeedRandom` is implemented, but it MAY reject external seeds, returning `CKR_RANDOM_SEED_NOT_SUPPORTED` depending on backend setup. Due to internal, hardware-assisted seeding, which is always used if available., one can not synchronize multiple modules RNG states. Due to this limitation, external seeding is mainly not useful in the context of EP11. It is only provided to allow hosts to influence, and therefore perturb, the state of modules without relying only on the module-internal entropy source.

Module-internal entropy sources are conditioned, and internally provided entropy is assumed to be full entropy of high quality. If external seeding is supported, the resulting mixed stream is assumed to preserve the higher entropy rate of the two streams.

This function always attempts seeding the backend, never filtering a valid-looking request. There are three mode-related expected results: accepted (CKR_OK), rejected due to general prohibition (CKR_RANDOM_SEED_NOT_SUPPORTED), rejected due to policy (general CP/policy rejection). Other failures are as applicable.

```
CK_RV m_SeedRandom (CK_BYTE_PTR pSeed, CK_ULONG ulSeedLen, target_t target) ;
```

m_SetAttributeValue

Implementation of PKCS#11 `C_SetAttributeValue`.

attribute packing: see _GetAttrValue

Currently, we only send Boolean attributes, all other attributes are handled by host (and we don't let modify arrays, such as WRAP_TEMPLATE).

Does not represent/need sessions (part of blob), therefore does not use the PKCS11 `hSession` parameter.

```
CK_RV m_SetAttributeValue (unsigned char *obj,        size_t olen,
                         CK_ATTRIBUTE_PTR pTemplate, CK_ULONG ulCount,
                                 target_t target) ;
```

m_shutdown

`m_shutdown` closes the AP device and resets the internal state of the host library and closes any [platform-specific] transport structures.

On non-production, single-binary builds, this function MAY also call backend-termination functions. No backend calls are made in setups where host and backend are disjunct, including HSM or socket-attached backends [i.e., calling `m_shutdown` has no effect on the backend in any production builds]

This function is not thread safe.

`XCP_DEVICE_ERROR` is returned if there was an problem closing the AP device.

```
int m_shutdown(void) ;
```

m_Sign

Implementation of PKCS#11 `C_Sign`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must map session to stored state.)

The `state` blob was output from: `SignInit`.

```
CK_RV m_Sign   (const unsigned char *state,     size_t stlen,
                        CK_BYTE_PTR data,     CK_ULONG dlen,
                        CK_BYTE_PTR sig,  CK_ULONG_PTR slen,
                           target_t target) ;
```

m_SignFinal

Implementation of PKCS#11 `C_SignFinal`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must map session to stored state.)

The `state` blob was output from: `SignInit`, `SignUpdate`.

```
CK_RV m_SignFinal (const unsigned char *state,      size_t stlen,
                          CK_BYTE_PTR sig,   CK_ULONG_PTR slen,
                             target_t target) ;
```

m_SignInit

Implementation of PKCS#11 `C_SignInit`.

```
CK_RV m_SignInit (unsigned char *state, size_t *slen,
              CK_MECHANISM_PTR pmech,
            const unsigned char *key,   size_t klen,
                       target_t target) ;
```

m_SignSingle

Nonstandard extension, combination of SignInit and Sign. Signs or MACs data in one pass, with one call, without constructing intermediate digest state. Does not return any state to host, only result.

This is the preferred way of signing, without an additional roundtrip and en/decryption. Functionally, SignSingle is equivalent to SignInit followed immediately by Sign.

The (key, klen) blob and the pmech mechanism together must be passable to SignInit.

Multi-data requests for HMAC and CMAC signatures are supported (sub-variants 2 and 3).

See also: SignInit, Sign, VerifySingle.

```
CK_RV m_SignSingle(const unsigned char *key,       size_t klen,
                     CK_MECHANISM_PTR pmech,
                          CK_BYTE_PTR data,     CK_ULONG dlen,
                          CK_BYTE_PTR sig,  CK_ULONG_PTR slen,
                             target_t target) ;
```

m_SignUpdate

Implementation of PKCS#11 C_SignUpdate.

The state,slen blob must be mapped from the PKCS11 hSession parameter. (Host library must map session to stored state.)

The state blob was output from: SignInit.

```
CK_RV m_SignUpdate(unsigned char *state,  size_t slen,
                   CK_BYTE_PTR data,  CK_ULONG dlen,
                      target_t target) ;
```

m_UnwrapKey

Implementation of PKCS#11 `C_UnwrapKey`.

`uwmech` specifies the encryption mechanism used to decrypt wrapped data. `ptempl` is a *key(pair)* parameter list, specifying how to transform the unwrapped data to a new key (must include `CKA_KEY_TYPE`; for others see `GenerateKey` and `GenerateKeyPair`).

The generated object is returned under `(unwrapped, uwlen)` as a blob. Symmetric keys return their key checksum (3 bytes) under `(csum, cslen)`; public-key objects return their public key as an SPKI in `(csum, cslen)`. Both forms are followed by a 4-byte big-endian value, encoding bitcount of the unwrapped key.

When transforming an SPKI to a MACed SPKI, one must use CKM_IBM_TRANSPORTKEY as the unwrapping mechanism. This mode supplies the raw SPKI as wrapped data, and ignores the KEK.

Note that `UnwrapKey` produces parity-adjusted DES keys (within the blobs), but tolerates input with improper parity.

```
CK_RV m_UnwrapKey (const   CK_BYTE_PTR wrapped,  CK_ULONG wlen,
                   const unsigned char *kek,       size_t keklen,
                   const unsigned char *mackey,    size_t mklen,
                   const unsigned char *pin,       size_t pinlen,
                const CK_MECHANISM_PTR uwmech,
                const CK_ATTRIBUTE_PTR ptempl,   CK_ULONG pcount,
                         unsigned char *unwrapped, size_t *uwlen,
                           CK_BYTE_PTR csum,     CK_ULONG *cslen,
                              target_t target) ;
```

m_Verify

Implementation of PKCS#11 `C_Verify`.

Does not update `(state, slen)`.

Note that the relative order of data and signature are reversed relative to `VerifySingle`.

The `state,slen` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must map session to stored state.)

The `state` blob was output from: `VerifyInit`.

```
CK_RV m_Verify (const unsigned char *state,     size_t stlen,
                        CK_BYTE_PTR data,     CK_ULONG dlen,
                        CK_BYTE_PTR sig,      CK_ULONG slen,
                           target_t target) ;
```

m_VerifyFinal

Implementation of PKCS#11 `C_VerifyFinal`.

Does not update `(state, slen)`.

The `state`,`slen` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must map session to stored state.)

The `state` blob was output from: `VerifyInit`, `VerifyUpdate`.

```
CK_RV m_VerifyFinal(const unsigned char *state, size_t stlen,
                            CK_BYTE_PTR sig,  CK_ULONG siglen,
                               target_t target) ;
```

m_VerifyInit

Implementation of PKCS#11 C_VerifyInit.

Given a key blob (key, klen), initialize a verify session state in (state, slen). The key blob may be a public key object, or HMAC key bytes. Key blob type must be consistent with pmech.

For public-key mechanisms, (key, klen) must contain an SPKI. This SPKI may be MACed (such as returned earlier by GenerateKeyPair) or just the SPKI itself (if obtained from an external source, such as a certificate).

If initializing an HMAC operation, session restrictions of the Verify object are inherited from the HMAC key. Since SPKIs are not tied to sessions, public-key Verify states are session-free.

The key,klen blob must be mapped from the PKCS11 hKey parameter.

Note: SignInit and VerifyInit are internally the same for HMAC and other symmetric/MAC mechanisms, other than enforcing different restrictions different restrictions (sign and verify, respectively).

```
CK_RV m_VerifyInit (unsigned char *state, size_t *slen,
                 CK_MECHANISM_PTR pmech,
              const unsigned char *key,   size_t klen,
                          target_t target) ;
```

m_VerifySingle

Nonstandard extension, combination of VerifyInit and Verify. Signs or MACs data in one pass, with one call, without constructing intermediate digest state. Does not return any state to host, only verification result. There is no size query, since this function returns a Boolean.

This is the preferred way of verifying a signature, without an additional roundtrip and en/decryption. Functionally, VerifySingle is equivalent to VerifyInit followed immediately by a Verify.

The (key, klen) blob and the pmech mechanism together must be passable to VerifyInit.

For public-key mechanisms, (key, klen) must contain an SPKI. This SPKI may be MACed (such as returned as a public key from GenerateKeyPair) or just the SPKI itself (if obtained from an external source, such as a certificate).

See also: VerifyInit, Verify, SignSingle.

```
CK_RV m_VerifySingle (const unsigned char *key,   size_t klen,
                          CK_MECHANISM_PTR pmech,
                               CK_BYTE_PTR data, CK_ULONG dlen,
                               CK_BYTE_PTR sig,  CK_ULONG slen,
                                  target_t target) ;
```

m_VerifyUpdate

Implementation of PKCS#11 `C_VerifyUpdate`.

The `state,slen` blob must be mapped from the PKCS11 `hSession` parameter. (Host library must map session to stored state.)

The `state` blob was output from: `VerifyInit`.

```
CK_RV m_VerifyUpdate(unsigned char *state,  size_t slen,
                       CK_BYTE_PTR data,  CK_ULONG dlen,
                          target_t target) ;
```

m_wire

Nonstandard extension: take constructed payload, send to target, return raw response.

Returns CKR_OK if response has been received. If the response contains a properly formed response, its internal response code is written to *irv.

This command has no function ID of its own. It inherits parameter count and syntax from the request we are passing to the backend.

```
CK_RV m_wire (unsigned char *wrsp, size_t *rsplen, CK_RV *irv,
        const unsigned char *req,  size_t reqlen,
               unsigned int flags,
                   target_t target) ;
```

m_WrapKey

Implementation of PKCS#11 `C_WrapKey`.

```
CK_RV m_WrapKey (const unsigned char *key,    size_t keylen,
                 const unsigned char *kek,    size_t keklen,
                 const unsigned char *mackey, size_t mklen,
              const CK_MECHANISM_PTR pmech,
                         CK_BYTE_PTR wrapped, CK_ULONG_PTR wlen,
                            target_t target) ;
```

Bit	*Name*	*Notes*
1	`XCP_BLOB_EXTRACTABLE`	May be encrypted by other keys. May not be reset.
2	`XCP_BLOB_NEVER_EXTRACTABLE`	set if key was created non-extractable. Set only initially, may not be modified.
4	`XCP_BLOB_MODIFIABLE`	attributes may be changed
8	`XCP_BLOB_NEVER_MODIFIABLE`	object was created read-only. Set only initially, may not be modified.
0x10	`XCP_BLOB_RESTRICTABLE`	attributes may be removed, but may not be made more permissive.
0x20	`XCP_BLOB_LOCAL`	was created inside this CSP, was not imported. Set upon object creation, may not be modified.
0x40	`XCP_BLOB_ATTRBOUND`	may be transported only in attribute-bound formats, but not pure PKCS11 ones. May not be modified.
0x80	`XCP_BLOB_USE_AS_DATA`	raw key bytes may be input to other processing as data, such as hashed, or deriving keys from them.
0x0100	`XCP_BLOB_SIGN`	may generate signatures
0x0200	`XCP_BLOB_SIGN_RECOVER`	may generate (asymmetric) signatures with message recovery
0x0400	`XCP_BLOB_DECRYPT`	may decrypt data
0x0800	`XCP_BLOB_ENCRYPT`	may encrypt data
0x1000	`XCP_BLOB_DERIVE`	may derive other keys
0x2000	`XCP_BLOB_UNWRAP`	may decrypt (transport) other keys
0x4000	`XCP_BLOB_WRAP`	may encrypt (transport) other keys
0x8000	`XCP_BLOB_VERIFY`	may verify signatures
0x010000	`XCP_BLOB_VERIFY_RECOVER`	may verify signatures and recover signed messages
0x020000	`XCP_BLOB_TRUSTED`	PKCS11 CKA_TRUSTED key
0x040000	`XCP_BLOB_WRAP_W_TRUSTED`	needs CKA_TRUSTED KEK note: _TRUSTED enforcement does not provide security guarantees. We only track it inside the HSM to assist hosts.
0x080000	`XCP_BLOB_RETAINED`	blob resides within backend, not (no longer) on host
0x100000	`XCP_BLOB_ALWAYS_RETAINED`	blob has been generated inside
0x200000	`XCP_BLOB_PROTKEY_EXTRACTABLE`	May be imported as protected key. May not be reset.
0x400000	`XCP_BLOB_PROTKEY_NEVER_EXTRACTABLE`	set if key was created non-extractable as a protected key. Set only initially, may not be modified.
...	`XCP_BLOB_BIT_MAX`	

Table 9: Blob attributes (usage restriction bits)

Bit	*Name*	*Notes*
0	XCP_CPB_ADD_CPBS	allow addition (activation) of CP bits
1	XCP_CPB_DELETE_CPBS	disable activating further control points (remove both ADD_CPBs and DELETE_CPBs to make unit read-only)
2	XCP_CPB_SIGN_ASYMM	sign with private keys
3	XCP_CPB_SIGN_SYMM	sign with HMAC or CMAC
4	XCP_CPB_SIGVERIFY_SYMM	verify with HMAC or CMAC. No asymmetric counterpart: one may not restrict use of public keys.
5	XCP_CPB_ENCRYPT_SYMM	encrypt with symmetric keys. No asymmetric counterpart: one may not restrict use of public keys.
6	XCP_CPB_DECRYPT_ASYMM	decrypt with private keys
7	XCP_CPB_DECRYPT_SYMM	decrypt with symmetric keys
8	XCP_CPB_WRAP_ASYMM	key export with public keys
9	XCP_CPB_WRAP_SYMM	key export with symmetric keys
10	XCP_CPB_UNWRAP_ASYMM	key import with private keys
11	XCP_CPB_UNWRAP_SYMM	key import with symmetric keys
12	XCP_CPB_KEYGEN_ASYMM	generate asymmetric keypairs (fn:GenerateKeyPair)
13	XCP_CPB_KEYGEN_SYMM	generate or derive symmetric keys including DSA or DH parameters
14	XCP_CPB_RETAINKEYS	allow backend to save semi-retained keys
15	XCP_CPB_SKIP_KEYTESTS	disable selftests on new asymmetric keys
16	XCP_CPB_NON_ATTRBOUND	allow keywrap without attribute-binding
17	XCP_CPB_MODIFY_OBJECTS	allow changes to objects (Booleans only)
18	XCP_CPB_RNG_SEED	allow mixing external seed to RNG backend may restrict further
19	XCP_CPB_ALG_RAW_RSA	allow RSA private-key use without padding (highly discouraged)
20	XCP_CPB_ALG_NFIPS2009	allow non-FIPS-approved algs (as of 2009) including non-FIPS keysizes
21	XCP_CPB_ALG_NBSI2009	allow non-BSI algorithms (as of 2009) including non-BSI keysizes
22	XCP_CPB_KEYSZ_HMAC_ANY	don't enforce minimum keysize on HMAC (allows keys shorter than half of digest)
23	XCP_CPB_KEYSZ_BELOW80BIT	allow algorithms below 80-bit strength
24	XCP_CPB_KEYSZ_80BIT	allow 80 to 111-bit algorithms
25	XCP_CPB_KEYSZ_112BIT	allow 112 to 127-bit algorithms
26	XCP_CPB_KEYSZ_128BIT	allow 128 to 191-bit algorithms
27	XCP_CPB_KEYSZ_192BIT	allow 192 to 255-bit algorithms
28	XCP_CPB_KEYSZ_256BIT	allow 256-bit algorithms
29	XCP_CPB_KEYSZ_RSA65536	allow RSA public exponents below 0x10001
30	XCP_CPB_ALG_RSA	RSA private-key or key-encrypt use
31	XCP_CPB_ALG_DSA	DSA private-key use
32	XCP_CPB_ALG_EC	EC private-key use (see CP on curves)
33	XCP_CPB_ALG_EC_BPOOLCRV	Brainpool (E.U.) EC curves
34	XCP_CPB_ALG_EC_NISTCRV	NIST/SECG EC curves
35	XCP_CPB_ALG_NFIPS2011	allow non-FIPS-approved algs (as of 2011) including non-FIPS keysizes
36	XCP_CPB_ALG_NBSI2011	allow non-BSI algorithms (as of 2011) including non-BSI keysizes
37	XCP_CPB_USER_SET_TRUSTED	allow non-admin set TRUSTED on blob/SPKI
38	XCP_CPB_ALG_SKIP_CROSSCHK	do not double-check sign/decrypt ops
39	XCP_CPB_WRAP_CRYPT_KEYS	allow keys which can en/decrypt data and also un/wrap other keys (applies to both generation and use)
40	XCP_CPB_SIGN_CRYPT_KEYS	allow keys which can en/decrypt data and also sign/verify (applies to both generation and use)
41	XCP_CPB_WRAP_SIGN_KEYS	allow keys which can un/wrap data and also sign/verify (applies to both generation and use)
42	XCP_CPB_USER_SET_ATTRBOUND	allow non-administrators to mark public key objects ATTRBOUND
43	XCP_CPB_ALLOW_PASSPHRASE	allow host to pass passprases, such as PKCS12 data, in the clear
44	XCP_CPB_WRAP_STRONGER_KEY	allow wrapping of stronger keys by weaker keys
45	XCP_CPB_WRAP_WITH_RAW_SPKI	allow wrapping with SPKIs without MAC and attributes
46	XCP_CPB_ALG_DH	Diffie-Hellman use (private keys)
47	XCP_CPB_DERIVE	allow key derivation (symmetric+EC/DH)
55	XCP_CPB_ALG_EC_25519	enable support of curve25519, c448 and related algorithms incl. EdDSA (ed25519 and ed448)
61	XCP_CPB_ALG_NBSI2017	allow non-BSI algorithms (as of 2017) including non-BSI keysizes (fn:Sign/RSA)
64	XCP_CPB_CPACF_PK	support data key generation and import for protected key
65	XCP_CPB_ALG_PQC	support for PQ algorithms (top CPB)
...	XCP_CPBITS_MAX	

Table 11: Control points

```
wire format

  1.  Requests
      1.1.  Function identifier
      1.1.1.  API ordinal number
      1.1.2.  API query
      1.2.  Size query
      1.3.  Request headers
      1.3.1.  CPRB [Coprocessor request block] wire headers
      1.4.  ASN.1 encoding
      1.4.1.  Multi-data requests
      1.4.2.  Restrictions of multi-data processing
      1.4.3.  Multi-data responses
      1.4.4.  Multi-data size queries
      1.4.5.  Multi-data error handling
  2.  Responses
      2.1.  Size query
      2.2.  Response headers
      2.3.  ASN.1 encoding
  3.  Key objects (blobs)
      3.1.  Raw keys
      3.1.1.  Blob revision
      3.2.  Blob attributes
      3.2.1.  Attribute header
      3.2.2.  Boolean attributes
      3.2.3.  Integer attributes
      3.2.4.  Variable-length attributes
      3.3.  Attribute-bound (AB) objects
      3.3.1.  AB transport header
      3.3.2.  AB encryption
      3.3.3.  AB signatures
      3.4.  PIN blobs
  4.  Administrative services
      4.1.  Administrative requests
      4.2.  Administrative responses
      4.3.  Administrative command block
      4.3.1.  Certificate replacement
      4.4   Administrative response block
      4.5.  Administrative structures
      4.5.1.  Integer administrative attributes
      4.5.2.  Control points (CPs)
  5.  Compound structures
      5.1.  Query types
      5.1.1.  Module query
      5.1.1.1.  Text/description fields
      5.1.2.  Domains query
      5.1.3.  Domain query
      5.1.4.  Selftest
      5.1.5.  Audit records history
      5.1.6.  Host queries
      5.1.7.  Performance classification statistics
      5.2.  Development test structures & calls
      5.2.1.  Set current WK
      5.2.2.  Set pending WK
      5.2.3.  Set control points (CPs)
```

```
5.2.4.  Iterated encryption/decryption
5.2.5.  Manage read-only mode
5.2.6.  Blob and CSP configuration query
5.2.6.1.  Blob configuration flags
5.2.6.2.  Blob object header configuration
5.2.7.  Algorithm tests
5.2.7.1.  RSA key generation (ANSI x9.31)
5.2.7.2.  DSA parameter (PQG) generation
5.2.7.3.  DSA PQG parameters, response
5.2.7.4.  EC scalar multiply, looped
5.2.8.  Performance-test events
5.2.8.1.  Performance-test measurements
5.2.8.2.  Host-influenced delay (NOP)
5.2.9.  Performance of raw primitives
5.2.10.  Blob-cache configuration
5.2.11.  Set/query FCV
5.3.  Serialized module state
5.3.1.  Tags within module state sub-types
5.3.1.1.  Request to export
5.3.1.2.  Request to export non-sensitive state
5.3.1.3.  Request to import into multi-domain setup
5.3.1.4.  Serialized state
5.3.1.5.  Exported keyparts
5.3.1.6.  Keyparts for import
5.3.2.  File-content encryption
5.3.2.1.  File-content encryption algorithms
5.3.3.  Lack of instance identifiers
5.3.4.  Keypart holder certificates
5.3.5.  State import/export scope restrictions
5.3.5.1.  Domain-restricted sections
5.3.5.2.  Non-sensitive restricted sections
5.3.6.  Verification during state import
5.4.  Audit records
5.4.1.  Audit record identification
5.4.2.  Undefined audit record timestamps
5.4.3.  Audit event types
5.4.4.  Audit key-records
5.4.4.1. Audit entries for key sizes
5.4.5.  Audit event flags
5.4.6.  Audit record salting
5.4.6.1.  Deterministic audit-record salt
5.4.7.  Signed audit records
5.4.8.  Audit record queries
5.4.8.1.  Audit history size
5.4.8.2.  Audit history entry
5.5.  Structures used by extensions
5.5.1.  Multi-level security (MLS) memory-typing extensions
5.5.2.  TRNG entropy-pool log access

5.6. Host target and module management
5.6.1 Target token data type
5.6.2 The XCP_Module structure
5.6.2.1 Structure version
5.6.2.2 Module flags
5.6.2.3 Domains and domainmask
5.6.2.4 The module number
5.6.2.5 The socket field
5.6.2.6 The module handle
5.6.2.7 The perf field
5.6.3. Flag dependent host library behavior
```

5.6.3.1 Fail on correctable errors
5.6.3.2 Virtual modules
5.6.3.3 Backend types
5.6.3.4 On the fly target adding
5.6.3.5 Probing targets
5.6.3.6 Performance measurements
5.6.4 Correctable and uncorrectable errors
5.6.4.1 Backend handling errors
5.6.4.2 Probing unreachable targets
5.6.4.3 Domain index correction
5.6.5 Host specific return codes
6. Primitive structures
6.1. Source partition/VM
6.2. Target domain
6.2.1. Administrative domain
6.2.2. Session identifier
6.3. Module identifier
6.4. Administrative transaction counter
6.4.1. Transaction counter queries
6.5. Imported keyparts
6.5.1. Imported raw keyparts, without reassembly threshold
6.5.2. Import of N-of-M keyparts
6.6. Exported keyparts
6.6.1. Exported raw keyparts
6.7. Key verification patterns (VPs)
6.7.1. Wrapping key identifier (WKID)
6.8. PKCS#11 structures
6.8.1. Mechanisms
6.8.1.1. Mechanisms with IV
6.8.1.2. RSA-PSS and OAEP
6.8.1.3. Custom ID-card (EAC) related functions
6.8.1.3.1. EAC sub-variants
6.8.1.4. ECDH1_DERIVE
6.8.1.5. CKM_IBM_CMAC
6.8.1.6. DSA parameters
6.8.1.7. PBE parameters (CKM_PBE_... mechanisms)
6.8.1.8. SPKI MAC conversion (CKM_IBM_TRANSPORTKEY)
6.8.1.9. DH parameters
6.8.1.10. DH key derivation
6.8.1.11. Mechanisms with variable-sized results (..._GENERIC)
6.8.1.12. Protected key import
6.8.1.19. Post-quantum algorithms: CKM_IBM_DILITHIUM variants
6.8.1.20. Montgomery and Edwards based elliptic curves
6.8.2. Mechanism list
6.8.3. Attributes
6.8.3.1. Compliance attributes
6.9. Raw integers
6.10. Empty fields
6.11. Date/time fields
6.11.1. Incremental date/time
6.11.2. Audit record date/time
6.11.3. Performance-test date/time
6.12. Certificate index
6.13. Query structures
6.13.1. Extended flags
6.13.2. Domain flags
6.13.3. Function Control vector (FCV)
6.14. Firmware identifier (FWID)
6.15. Domain mask
6.15.1. Zero-based domain mask

6.15.2. Domain mask window
6.16. Retained key identifier (RKID)
6.16.1. Truncated Retained key identifier
6.17. File-parts and their identification
6.17.1. File-size query
6.17.2. Setting file size
6.18. Bitmasks
7. Other standard formats
7.1. SignerInfo
7.2. RecipientInfo
7.2.1. EC RecipientInfo
7.3. SubjectKeyIdentifier (SKI)
7.3.1. Lists of SKIs
7.4. SubjectPublicKeyInfo (SPKI)
7.4.1. SPKI with MAC
7.4.2. SPKIs for curve25519, ed25519, and c448 variants
7.4.3. PKCS8 private keys for curve25519, ed25519, and c448 variants
7.4.3.1. Sample structures for curve25519, ed25519, and c448 variants
7.5. Certificates
7.6. PKCS#11 key checksums
7.6.1. Key checksums of PBE mechanisms
7.6.2. Public-key types' checksums
7.7. ASN.1/BER TLV (tag, length, value) encoding
7.7.1. Requests
7.7.2. Responses
7.8. DSA/DH parameters
7.8.1. Raw DSA/DH P,Q,G parameters
7.8.2. Raw DH P,G parameters
7.9. Private-key objects
8. Constants
8.1. Administrative identifiers
8.1.1. Administrative variables (attributes)
8.1.1.1. Permissions
8.1.1.1.1. Non-modifiable permissions
8.1.1.2. Infrastructure mode
8.1.1.3. Operational mode (compliance settings)
8.1.1.3.1. FIPS2009 mode
8.1.1.3.2. FIPS2011 mode
8.1.1.3.3. BSI/2009 mode
8.1.1.3.4. BSI/2011 mode
8.1.1.3.5. Common Criteria Certification mode
8.1.1.4. Importer keytypes
8.1.1.5. EC curve identifiers
8.1.1.5.1. Non-Weierstrass EC curve identifiers
8.1.2. Serialized state types (tags)
8.1.3. File identifiers
8.2. Function identifiers
8.2.1. Function (performance) category identifiers
8.3. Card control points
8.4. Domain control points (CPs)
8.5. Blob Boolean attributes
8.5.1. Vendor-defined attributes (CKA_IBM_...)
8.6. Test mechanism constants
8.6.1. RNG test constants
8.6.2. Blob-cache control
8.7. Other fixed values
8.7.1. Query types
8.7.1.1. Query types: extended capabilities
8.7.1.2. Query types: reserved host types
8.7.1.3. Target support bitmask: Supported target modes of host library

```
      8.7.1.4.  Query type: elliptic curve support
      8.7.1.5.  Query type: elliptic curve group support
      8.7.2.  Return values (CKR)
      8.7.3.  Custom mechanisms (CKM) and related constants
      8.7.4.  Attribute types reported by keytype/mechanism-attribute query
      8.8.  Referenced PKCS#11 constants
      8.9.  Audit-related constants
      8.9.1.  Audit event types
      8.9.2.  Specific audit events
      8.9.3.  Audit event flags
      8.10.  Function sub-variants
  9.  Command parameter lists
      9.1.  Key generation
      9.2.  Digesting
      9.3.  Signing and verification
      9.4.  Encryption and decryption
      9.5.  Random numbers
      9.6.  Key transport
      9.7.  Queries
      9.8.  Administration
      9.8.1.  Session management
      9.9.  Other calls
 10.  Version dependent changes
      10.1.  GetMechanismInfo - HMAC min/maxKeySize
      10.2.  API ordinal
      10.2.1. API ordinal 3 - DeriveKey
      10.2.2. API ordinal 4 - Protected key import
```

The following sections contain fixed values such as sizes. Where relevant, we also list the corresponding constant name from the backend source, possibly also exposed through the official host header (ep11.h). Host code directly producing/parsing wire packets would not be aware of our symbolic names, but those utilizing the host C library, or those working on backend could still correlate them.

1. Requests

Requests are serialized in the following ASN.1 form:

```
xcpReq ::= SEQUENCE {
    functionId      OCTET STRING,     -- (1.1)
    domain          OCTET STRING,     -- raw domain (6.2)
                                      -- ignored for card-level
                                      -- administrative actions
    parameter/s     OCTET STRING...
}
```

The function identifier (1.1) selects the PKCS#11 or xcp top-level function which must parse the request. This field must remain the first one, to allow the receiver to recognize the function, and find the required number of parameters which follow.

The domain field (6.2) must match the domain field of the encapsulating request header, if that also contains domain targeting information (such as with CPRBs, in (1.3.1)). For card-level administrative requests the domain field of the encapsulating request header (if any) is ignored.

The number of parameters is command-specific. Non-administrative commands without input parameters---queries---contain only functionId and domain. Administrative commands always contain input parameters at the outer xcpReq, as their requests are packaged in separately constructed administrative command blocks, within regular parameter/s, see (4.1)

1.1. Function identifier

Functions combine an API ordinal number (2 bytes, big-endian) (see also 1.1.1) and a function ordinal number (2 bytes, big-endian) into a fixed-size, 32-bit value. The function ordinal selects the PKCS#11 function, or xcp-specific additions. Function ordinals are expected to persist across future API updates; backends may provide compatibility modes, if the caller asks for previous API revisions (through the API ordinal).

The list of valid function identifiers is in (8.2).

Sub-variants of functions are encoded as a separate parameter (the first one), where appropriate. Sub-variant selection, where supported, is listed under function-specific parameters (9). Size queries, as the most frequently used sub-variant, are described separately (1.2)

1.1.1. API ordinal number

The API ordinal number, a 16-bit integer, tracks host-visible revisions of backend services. It gets incremented at each API-changing release. Host code may adapt to changing API ordinals, depending on what is being changed. Ordinal 0 is reserved, and will not be used except in noted exceptional cases.

The API ordinal, included in function identifier fields, allows the backend to implement past versions in API compatibility mode. Requests from the future---an ordinal number beyond that of the backend---are rejected.

See section 5.6.2.8 for more details on how the API ordinal can be used and determined.

The current API ordinal is 0x0004.

If the API ordinal reaches 0xffff, an escape mechanism will be used to represent a longer API ordinal. Note that we envision further changes before we expect to reach the API ordinal limit of 0xffff.

1.1.2. API query

The API version query returns the following summary structure:

```
xcpRsp ::= SEQUENCE {
    API version          OCTET STRING,
            -- 4 bytes: API major version, minor version, 2x 00
    reserved             OCTET STRING,
    build configuration  OCTET STRING  -- truncated to 4 bytes
}
```

The reserved fields are included for a future addition, reporting the range of functions available, and the range of available function sub-variants.

1.2. Size query

PKCS#11 supports size queries, where a reasonable, potentially conservative estimate of the output size must be returned, without performing the actual operation. XCP supports size queries as a subset of function variants: the first non-default function variant (1) is reserved for size query.

When querying output sizes, in most cases the meaning of input and output _data_ changes: size queries pass input _bytecount_ as ‘‘input data’’ in a fixed-size raw integer field (8 bytes), instead of actual input data. Parameter count, and by implication, wire encoding, is unchanged. Function descriptions (9) specify the special cases where a size query does not change the meaning of data fields to bytecount.

Similarly, size queries return the output _bytecount_, encoded as a raw integer of 8 bytes as ‘‘output data’’ (2.1).

The meaning of input _data_ fields only changes for actual user data. Size queries where no user data is used, such as key generation or state--session--initialization based on input blobs, any blobs must still be passed to the backend.

Not all functions support size queries. Exceptions include functions with fixed-size output--either user-specified sizes, or known structures of fixed size--and are unambiguously documented. Host libraries must handle PKCS#11-compliant reporting of size queries for such cases.

Note that encoding bytecounts instead of data actually increases request size in pathological cases, such as when digesting small numbers of bytes.

1.3. Request headers

Transport-specific headers precede request/response payloads (SEQUENCEs), and are system-dependent.

1.3.1. CPRB [Coprocessor request block] wire headers

On HSM-resident mainframe backends, we use a fixed CPRB header layout to target requests. The format is identical for requests and responses:

	offset		bytes	note
1.	0	CPRB (header) bytecount	2	fixed 32 (X’0020)
2.	2	CPRB version	1	0x04
3.	3	reserved	1	0
4.	4	reserved	1	0 incoming, performance when returned
5.	5	flags	1	reserved bits:
				x80 OS(0) or TKE/administration(1)
				x40 EP11(0) or Miniboot(1)
				[hw-Miniboot uses the same head]
				x20 protected key import cmd
				[to be intercepted by firmware]
				other bits are reserved 0
6.	6	CPRB subtype	2	X’5434 (ASCII ‘‘T4’’)
7.	8	source identifier	4	partition/VM identifier (6.1)
8.	12	target domain	4	raw domain (6.2)

```
9.   16   return code            4      reserved 0 for requests
10.  20   reserved               8      reserved 0
11.  28   payload bytecount      4      net bytecount of following SEQUENCE
     -------------------------------------------------------------------------
     32   (payload)              (var)  (see bytecount above)
```

Multi-byte fields are big-endian.

The source identifier refers to the originating partition/VM (see 6.1). It is currently ignored by the backend. Host code is responsible for matching targeting, i.e., verifying that the given source is allowed to send a request to the target domain. THE BACKEND DOES NOT VERIFY THE SOURCE IDENTIFIER.

The domain field (see 6.2) must match that within the request/response structure (within payload).

The SEQUENCE following a CPRB must be appended without intermediate padding. Payload bytecount is net size, ignoring any channel-mandated padding---if any---which should not be XCP-visible.

1.4. ASN.1 encoding

The backend accepts BER-encoded requests with definite encoding. It does not mandate DER encoding at the top-level SEQUENCE. Embedded content, such as the administrative command block (4.3), is DER-encoded.

See also (2.3) and (7.7) for restrictions on encoded data.

1.4.1. Multi-data requests

Certain functions support a sub-variant, which supplies data for multiple, unrelated operations to be processed in a single request. A typical application is signature generation on multiple, small inputs using the same key, amortizing I/O overhead over a batch of individually fast operations. Only data is interpreted differently for multi-data requests: all other conditions, such as key or mechanism (parameters) must be identical for these calls.

Input data to the multi-data request sub-variant, and its size-query equivalent, must be encoded within the data field which would otherwise supply the single input. Data sub-fields must be embedded within an ASN.1-encoded SEQUENCE of OCTET STRINGS, all within the OCTET STRING request-data field. The top-level encoding, or any of the other fields are unaffected; the only difference is the enforced internal structure of the data field.

As with other fields, presence of an empty OCTET STRING is allowed: it indicates 0-byte data in that position.

Note that multi-data request input sub-fields are processed in isolation: no capabilities for referencing other sub-fields etc. exist. Therefore, multi-data requests are functionally identical to multiple, single-data calls to the same service, and only provide the equivalent of small, local loops without additional capabilities.

1.4.2. Restrictions of multi-data processing

Multi-data requests are only supported by calls which do not need to update object or key state during processing. This restricts multi-data to single-pass forms of functional calls; generally, to PKCS11 single-pass nnn() [after nnnInit()] and our proprietary chained form, nnnSingle(). (In other words, only calls which do not return updated state to the host may be eligible for multi-data requests.) Since the transport itself may restrict request sizes, and therefore prevent passing oversized requests in single calls, further size limits may be imposed on multi-data request/response flows, which are not described here.

Backends MAY restrict the number of sub-data fields accepted for each multi-data request; this upper limit is reported under extended capabilities (8.7.1.1.) Backends restricting the number of fields MUST support at least 16 of them. Since the extended-capability query allows reporting ‘‘no predefined limit’’ as a non-zero value, this extended-capability field is the recommended query mechanism to detect the presence of multi-data support.

In their current form, multi-data requests may not support non-streaming operations which still require different states, such as symmetric MAC calculation with per-data IVs (as IVs in PKCS11 become part of the session state, not supplied with data). A future, compatible addition is planned for relevant modes, but it is not currently available in production.

1.4.3. Multi-data responses

Responses to multi-data requests are serialized as regular SEQUENCEs of OCTET STRINGS, all encapsulated by the OCTET STRING field of the response where output data is returned. The number of embedded OCTET STRING sub-fields and their relative order matches that supplied in the request SEQUENCE.

Responses contain no other data-identifying information, therefore callers must match response fields to the original inputs which were aggregated into a single multi-data request.

1.4.4. Multi-data size queries

Functions supporting multi-data requests, unless otherwise noted, provide size queries. Multi-data size queries must be supplied replacing the corresponding data field as described in (1.2), as a packed array of fix-sized bytecounts; no additional formatting or padding is required. Responses contain a similarly packed array of response bytecounts.

1.4.5. Multi-data error handling

If data encoding or the packed sizes for multi-data (size-query) requests is invalid, the backend reports CKR_IBM_TRANSPORT_ERROR.

Multi-data requests do not return partial data, and supply only a single return value. If any of the constituent requests fails, the proper return value is supplied; responses of any preceding, successful sub-operations are discarded.

We do not currently support diagnostics to report which particular request slice caused the error.

Note that the backend MAY impose restrictions on the total runtime of multi-data requests, and is encouraged to return CKR_IBM_REQ_TIMEOUT if the requests would terminate beyond what may be appropriate. Details

```
and limits of such resource limits are implementation-defined.

2.  Responses

Responses are serialized in the following ASN.1 form, encoded as BER:

xcpRsp ::= SEQUENCE {
    functionId      OCTET STRING,
    domain          OCTET STRING,      -- raw domain (6.2)
    returnValue     OCTET STRING,
    response/s      OCTET STRING...    -- if present
}

The functionId and domain fields are copied from the originating request
without modification.

The returnValue field contains a 32-bit fixed-size raw integer, encoding
a CKR_... PKCS#11 or extended (vendor-defined) return value (see also:
8.7.2). If the value is not CKR_OK (i.e., 0), this is the last field,
no response follows.

Response/s contain output(s) of the command. They are missing if the
request failed (i.e., returnValue is not CKR_OK). The number of returned
fields is command-specific. For commands which only return status--such
as C_Verify()--response fields are missing.

2.1.  Size query

As described in (1.2), most size queries return output bytecount as
fixed-size, 64-bit raw integers in the field where output---data---would
appear. Since we only change the interpretation of the "output" field,
parameter count and encapsulation are unchanged.

2.2.  Response headers

On HSM-resident mainframe backends, the CPRB header layout is
identical to that of requests (1.3.1).
The CPRB return code is zero in almost all cases. The current two
exceptions are fundamental ASN.1 parse errors within the payload or
a mismatch between the CPRB specified domain and the domain field(s)
within the payload (see 6.2).

The CPRB return codes currently defined are:

XCP_M2H_DEFAULT_ERR         0x000c0001
XCP_M2H_DOMAIN_MISMATCH     0x000c0002
XCP_M2H_FW_BUSY             0x000c0003  -- firmware is busy, try again later
                                        -- (only relevant for protected
                                        -- key import)
XCP_M2H_FN_NOT_ALLW         0x000c0004  -- function is not allowed
                                        -- (only relevant for protected
                                        -- key import)

2.3.  ASN.1 encoding

Responses are encoded under BER rules for the top-level SEQUENCE, with
one specific special case: variable-length fields are encoded with at
least a two-byte Length entry (i.e., the shortest encoded length is 82
|| BE16(L) for a value of L bytes). Variable-length values over 0xffff
bytes use a minimal encoding for their Length entries.
```

Integers which by construction fit within 32 bits--also described individually below--are returned as fixed-size OCTET STRINGS, therefore follow a fixed 2-byte Tag+Len field (hex 0404).

Individual content, such as administrative response blocks (4.2) are DER-encoded.

See also (7.7).

3. Key objects (blobs)

3.1. Raw keys

Individual keys are stored as encrypted, authenticated binary blobs on the host.

WK fields are concatenated without padding in the following order:

	bytecount	note
----- MACed: -----		
1. WK virtualization mask	32	session identifier (6.2.2) (XCP_WK_BYTES)
2. WK identifier	16	XCP_WKID_BYTES, see (6.7.1)
3. Boolean attributes	8	mirrored from attributes in encrypted region
4. mode identification	8	see (6.8.3.1) and (8.1.1.3)
----- IV -----		
5. structure version	2	see (3.1.1)
6. IV	16 -2	
		total 16 bytes (MOD_WRAP_BLOCKSIZE)
----- encrypted: -----		
7. attributes' bytecount	2	first encrypted field see MOD_VARLENS_BYTES
8. payload bytecount	2	see MOD_VARLENS_BYTES
9. auxiliary bytecount	2	see MOD_VARLENS_BYTES
10. reserved	2	mandatory zero-filled
----- blob fields -----		
11. attributes	(dynamic)	see bytecount above
12. serialized CSP object	(dynamic)	see bytecount above
13. auxiliary CSP data	(dynamic)	see bytecount above
14. padding, if needed	pad encrypted region to multiple of 16	skipped if not needed (unambiguous, based on starting bytecounts)
----- end of encrypted region -----		
----- end of MACed region -----		
15. MAC	32	covers all preceding bytes (incl. clear header)

WK virtualization mask is all-zero for objects not bound to sessions. Otherwise, the field contains the session identifier of the session the object is bound to. Contents of this field are combined with the controlling WK to derive a session-specific encryption key for objects bound to that session.

Blob Booleans and clear attributes must match their counterparts inside the encrypted region. They are replicated only to assist the host, namely to reduce the number of GetAttributeValue calls, and to allow

```
sorting of blobs based on operational mode.

The IV is formed from the blob version field (3.1.1) and a remaining,
module-originated random field.  These fields are concatenated to form
a single initialization vector (16, MOD_WRAP_BLOCKSIZE bytes).

Encrypted regions are assumed to be host-opaque. For test purposes, we
may construct/modify objects with known WKs.

Attribute and payload sizes are fixed-size raw integers. Blob attribute
encoding is in (3.2).

Details of serialized CSP objects are implementation-specific. (They
follow a proprietary, internal, de facto frozen interface, which is not
targeted or necessary for interoperability. Counterparties MUST use
PKCS11-standard Un/WrapKey to transport keys, and have no reason to
interface directly with non-key blobs.)

Padding, if present, zero-pads the encrypted region to an integer
multiple of MOD_OBJ_PADTO. Since bytecounts are unambigous--see
the first two fields of the encrypted region--we do not require an
unambiguously decodeable padding mode (such as PKCS padding). The
verifies that padding, if present, consists of all zeroes.

The MAC is calculated from a WK-derived MAC key, generating an
HMAC/SHA-256 signature on the preceding bytes. Objects are encrypted,
then MACed, including their clear header.

3.1.1.  Blob revision

Blob version is encoded as a single 16-bit identifier, stored as a
raw integer (at the fixed offset 64).  Current version is 0x1234.

3.2.  Blob attributes

Attributes share the same format within key blobs and queries.  In the latter
case, they are passed around in the clear.

3.2.1.  Attribute header

A full set of attributes concatenates a header, Booleans, integers,
and variable-length/array attributes without padding, in this order.
The header is a single 32-bit raw integer, concatenating the following
fields:

                                 bitcount    note
1.   version                     4           1  (fixed for current version)
2.   reserved                    4           0
     -----  byte 1:  --------------------------------------------------------
3.   number of integer attrs     8
     -----  bytes 2/3:  ------------------------------------------------------
4.   number of var-len attrs     4           variable-length/array
                                             attribute count
5.   wordcount of var-len attrs  12          measured in 32-bit words

The header is written as a big-endian buffer in MS to LS order, i.e.,
the first byte in the current version is fixed 0x10.

Variable-length/array attributes are always encoded to full words,
therefore their total size is measured in 32-bit units.  The current
```

format limits aggregate size to 131040 bytes ((2^12-1) * 4).

As described below, for an attribute field with N integer attributes and M variable-length ones, total field size is 12+8*N+L(M) bytes (4 /header/+ 2*4 /Booleans,two copies/ + 2*4*N /integers/ +L(M) /total of var-len fields' gross sizes/). Traversing variable-length attributes individually requires parsing; overall size of the field is available in encapsulations we use (therefore, the aggregate size of var-length fields should be available even without such parsing).

3.2.2. Boolean attributes

Boolean attributes are represented as a single big-endian, fixed-size integer, currently fixed at 32 bits, and are always present in an attribute structure. Individual bits are defined as XCP_BLOB_<nnn> constants (see 8.5). The field must not have any other bits set, other than the enumerated constants.

The wire form always includes two copies of Booleans. Except for ''attribute set'' requests, the two copies must be identical. We prefer to mandate a single--potentially redundant--form, with an additional check, less intrusive than having flexibility in the wire structure.

When attributes are being set (i.e., a C_SetAttributeValue request), the two Booleans may differ. The first value is the set mask (ORing all bits which must be modified), the second one is the actual value to set (containing 0 for those mask bits which will be reset to false).

3.2.3. Integer attributes

Integer attributes are limited to 32-bit values, with attribute-specific interpretation (generally, unsigned values). They are stored as a packed array of 32+32-bit big-endian integers, with the PKCS#11 attribute type (a CKA_... constant) followed by the 32-bit value.

The attribute header (3.2.4) contains the number of integer attributes, therefore the number of type+value pairs need not be encoded in the structure.

3.2.4. Variable-length attributes

Variable-length and array attributes are encoded as type-length-contents tuples, with type-dependent interpretation. The order is the following:

	bytecount	note
1. type	4	
2. length	4	net bytecount excluding content padding
3. contents	(length, round up to 4n)	zero-pad to multiple of 4, if applicable

Interpretation is attribute-dependent. Recursive attributes, such as CKA_WRAP_TEMPLATE or CKA_UNWRAP_TEMPLATE, contain full attribute structures as their content.

3.3. Attribute-bound (AB) key transport form

Note that the AB transport form is, unfortunately, incompatible with standard PKCS#11 formats. However, since none of the standard transport-encryption methods allow simultaneous transport of key material and its attributes, the standard formats clearly conflict with the requirements of high-assurance environments.

```
The AB transport form may be parsed in a single pass, containing
identifying metadata, information about the encryption key,
attributes, raw key bytes, and a signature, in this order.

     field                               bytes  notes
---  start of MACed/signed region  ------------------------------------------
1.   AB header                              16  XCP_AB_HDR_BYTES
2.   PK-encrypted transport key          <var>  present only if KEK is
                                                asymmetric key(pair); bytecount
                                                depends on KEK type and size
3.   IV                                <block>  block size of content-encryption
                                                cipher (8 or 16); see structure
                                                description below
---  start of symmetric-encrypted region  -----------------------------------
4.   attribute bytecount                     2  bytecount of field (6) below
5.   raw key structure bytecount             2  bytecount of field (7) below
6.   packed attributes              <variable>  full attribute structure, see
                                                notes below and (3.2)
7.   raw key bytes                  <variable>  layout differs for symmetric
                                                and private keys (3.3.2)
8.   padding, if necessary          <variable>  pads plaintext to multiple of
                                                32 (XCP_AB_PADTO), if necessary
---  end of symmetric-encrypted and MACed/signed region  -------------------
9.   MAC/digital signature of       <variable>  bytecount depends on KEK type
     preceding bytes                            and possibly size (3.3.3)

Attributes are packed as described in (3.2). The packed structure always
includes at least three integer attributes (3.2.2), with the first
three storing CKA_KEY_TYPE, CKA_VALUE_LEN, and the vendor attribute
CKA_IBM_STD_COMPLIANCE1 (0x8001000a), in this
order. If the original key had other integer attributes, they follow
these three in arbitrary, unspecified order.

If the size of encrypted fields is a multiple of AB padding size
(32 bytes), the (8) padding is missing. In other cases,
PKCS padding is applied, filling the padding field single bytes,
each containing the padding bytecount.

3.3.1.  AB transport header

Keys formatted for AB transport use a header of fixed-size
(16, XCP_AB_HDR_BYTES):

     offset
         field
                                  bitcount    note
-------------------------------------------------------------------------------
1.   0   format identifier               8    (see XCP_AB_HDR_ID)
2.   1   version/blocksize               8    currently, 0x11 or 0x12, see below
3.   2   packed length                  16    gross bytecount, in 8-byte units

4.   4   KEK checksum..................24    PKCS#11 checksum or MS bits of SKI
5.   7   key checksum                   24    PKCS#11 checksum or MS bits of SKI
6.  10   MAC/sign key checksum          24    PKCS#11 checksum or MS bits of SKI

7.  13   packed key bytecounts          24    MS 12 bits: encrypted transportkey
                                              LS 12 bits: MAC/dig.sig. bytecount
-------------------------------------------------------------------------------
    16   total header bytecount
```

Symmetric key checksums use standard 24-bit PKCS#11 checksums (the CKA_CHECK_VALUE attribute, as specified by PKCS#11 for the each key type). We do not currently support symmetric key types without a defined PKCS#11 check value---such as RC4---and therefore we only use standard checksums.

Asymmetric keys feature the most significant 24 bits of their SKI as checksum. By construction, the SKI is identical for public and private keys of the same keypair.

Note that the backend generates headers with proper checksums, when exporting. During import, key checksums are ignored. (Note that this won't impact security, as a rogue MAC-key owner can compromise the whole compound, including checksums; others can't benefit from changing only the checksums.)

3.3.2. AB encryption

The encryption used in AB transport formats depends on both KEK and key type. Since the format is self-describing---it describes the type and size of the embedded key before the encrypted field---and an improper KEK may be detected, AB encryption may be unambigously decoded.

If the KEK is symmetric, it encrypts the key directly, without an intermediate transport key. If the KEK is an asymmetric key(pair), encrypted regions are encrypted by a randomly generated 256-bit AES transport key, and the transport key is included in the AB-formatted header, encrypted by the KEK.

If the KEK is an RSA key(pair), the transport key is encrypted using RSA OAEP. AB transport does not currently support other asymmetric KEK types or transportkey-encryption algorithms.

Symmetric/secret keys are written into the leading bytes of a single, 256-byte buffer, with trailing bytes zero-filled. (Currently, we do not transport larger than 256-byte secret keys.) Since bytecount and key type are unambiguous---present in preceding attributes---the raw bytes require no further annotation or formatting.

Asymmetric keys are stored as PKCS#8 files, zero-padded to an integer multiple of 32 bits (4 bytes). The preceding length field (raw key structure bytecount) contains the unpadded bytecount.

The following keytypes are currently supported in AB transport:

	type	type(be32)	encoding	interpretation of CKA_VALUE_LEN
1	CKK_AES	0x0000001f	raw bytes	key size, bytes
2	CKK_DES	0x00000013	raw bytes	key size, bytes
3	CKK_GENERIC_SECRET	0x00000010	raw bytes	key size, bytes
4	CKK_RSA	0x00000000	PKCS#8	PKCS#8 bytecount
5	CKK_EC	0x00000003	PKCS#8	PKCS#8 bytecount
6	CKK_DSA	0x00000001	PKCS#8	PKCS#8 bytecount
7	CKK_DH	0x00000002	PKCS#8	PKCS#8 bytecount

3.3.3. AB signatures

Attribute-bound keys may be signed by both symmetric and asymmetric keys, producing a MAC or digital signature, respectively. Currently, the key type and size unambiguously determine the signing algorithm, and the AB form of keys does not specify algorithms. If algorithm choices are added in the future, they will be in the form of predefined selections from a fixed set.

If the signing/MAC key is symmetric, it is used as a MAC key, signing with HMAC/SHA-256 as a signature function.

If the signature key is an RSA key, the signature algorithm is RSA-PSS, with SHA-256 as a hash function.

If the signature key is an EC or DSA key, the signature algorithm is ECDSA or DSA, with SHA-256 as a hash function.

3.4. PIN blobs

PIN blobs store session IDs, MACed with potentially user-influenced salt. Fields of the PIN blob are concatenated without further formatting or padding:

	field	bytecount	note
1.	session identifier	32	(XCP_WK_BYTES)
2.	salt	16	(XCP_PIN_SALT_BYTES)
3.	MAC	32	(XCP_HMAC_BYTES) covers all preceding bytes (HMAC/SHA256)

PIN blobs are signed by a fixed, predefined, single-purpose HMAC key, independent of any WK-derived MAC keys, which is not considered to be sensitive. Host libraries are assumed to contribute sufficient separation into passphrases and nonces of their callers to prevent callers impersonating each other.

Salt construction forces a fixed prefix byte onto the salt field, and allows callers to influence, but not to fully specify, the salt, and therefore the PIN blob (see m_Login() parameters). Generated salt bytes may contain one of the following leading bytes, showing different types of caller-contributed nonces:

```
01 -- module-generated, all-random nonce, no caller-contributed data
      rest of salt is unstructured

02 -- caller-contributed nonce, >3 bytes, has been truncated to 3 bytes
03 -- caller-contributed nonce, <= 3 bytes, has been inserted verbatim
   -- both 02 and 03 subtypes store the following information in offsets 1..5:
      1 [ bytecount  ]    -- mod 256
      2 [ n0         ]    -- up to first 3 bytes of nonce
      3 [ n1         ]    -- unused bytes (for shorter nonces) are zero-filled
      4 [ n2         ]
      5 [ XOR(nonce) ]    -- XOR of all nonce bytes
```

There is potential ambiguity (collisions) when nonces over 3 bytes are supplied by callers. We do not consider these collisions problematic, and do not intend to change the scheme to prevent them, or to increase the effective size: we expect nonces to _contribute_ _some_ caller-supplied data, but do not expect this data to be unique.

Other leading salt bytes are reserved, and are not generated by current

```
firmware. In future versions, further values may be defined as part of
PIN-blob versioning.

Since PIN blobs are signed by the same key, they are portable across
modules: each receiving module can verify the HMAC on the concatenated
fields.

PIN blobs inherit construction and any restriction of session identifiers
they include (6.2.2).


4.  Administrative services

4.1  Administrative requests

Requests to administrative services, including queries and commands,
are encapsulated as regular command structures targeting the m_admin()
function. These requests contain an administrative command, and any
applicable services as parameters, therefore encoded as:

xcpAdminReq ::= SEQUENCE {
    functionId      OCTET STRING,      -- m_admin()
    domain          OCTET STRING,      -- raw domain (6.2)
                                       -- must match field within command block
    administrative  OCTET STRING encapsulates {
        command  xcpAdminBlk
    }
    signatures      OCTET STRING {
        -- signerInfo's, without encapsulating SET OF, if present
    }
}

See (4.3) for details of the administrative field, or (2) for an
overview of request/response encapsulation.

Signatures, if present, are encoded within ``signatures'', concatenating
raw SignerInfo structures (see 7.1). Each SignerInfo contains a
signature on the preceding ``administrative'' field. For requests
without signatures, the field must be present but empty. As a special
case, administrator logins---inserting new certificates---during
imprinting are also unsigned.

Signatures are calculated over the entire SEQUENCE of the
``administrative'' field including tag+length, but excluding its
encapsulating OCTET STRING.

4.2  Administrative responses

Administrative responses inherit their function identifier and domain
fields from the originating request.

xcpAdminRsp ::= SEQUENCE {
    functionId      OCTET STRING,      -- m_admin()
    domain          OCTET STRING,      -- raw domain (6.2)
                                      -- matches field of originating request
    returnValue     OCTET STRING,
    administrative  OCTET STRING encapsulates {
        response  xcpAdminRspBlk
    }
    signature       OCTET STRING {
        admSig    SignerInfo
    }
}
```

ReturnValue is defined to be compatible with regular responses; administrative commands return CKR_OK here except for infrastructure-level failures (and then administrative/signature fields are missing).

If returnValue contains CKR_OK, the actual return value within the administrative response block is applicable, which may be different from CKR_OK. (The reason for this distinction: logical failures should still return a meaningful, signed administrative response, this special case allows that. Otherwise, the non-CKR_OK return value would imply lack of further fields.)

The signature is calculated over the entire SEQUENCE of the ‘‘administrative’’ field including tag+length, but excluding its encapsulating OCTET STRING. The ‘‘response’’ structure is copied from the originating ‘‘command’’ structure, with only the payload field being replaced by the response payload.

4.3 Administrative command block

Command or query payloads contain fields to identify the command, targeted module and domain, transaction counter, and the payload--the latter interpreted in a service-specific way.

```
xcpAdminBlk ::= SEQUENCE {
    admFunctionId     OCTET STRING,   -- command/query identifier
    domain            OCTET STRING,   -- administrative domain (6.2.1)
                                      -- mandatory 0 for card-level actions
    moduleIdentifier  OCTET STRING,
    transactionCtr    OCTET STRING,   -- may be empty for queries
                                      -- or unsigned commands
    payload           OCTET STRING    -- command-specific interpretation
}
```

See (8.1) for command/query identifiers. The identifier is encoded as a fixed-size, 32-bit raw integer.

The domain must match that of the encapsulating request for domain-level actions. Note that within the command block, we use administrative domains (i.e., including domain instance identifier). For domain-level actions, the backend verifies that the--redundant--domain fields are consistent.

For card-level actions the domain field must be set to all-zeroes.

The module identifier, if present, restricts the command block to a particular backend (see 6.3). It may be omitted for queries, or commands without signatures; any host-supplied value is ignored in such a case. If the field is present, it must have the proper size.

The transaction counter (see 6.4) must supply the next transaction counter state for state-changing commands. The field may be empty for queries, or commands without signatures. If present in such a command block--with the proper size--the host-provided value is ignored.

4.3.1. Certificate replacement

When replacing a certificate, the payload must combine the SKI of the certificate being replaced, and the replacement certificate:

```
xcpCertReplacement ::= SEQUENCE {
    ski OCTET STRING,   -- SubjectKeyIdentifier
    rpl OCTET STRING    -- Certificate
}
```

This compound structure (see 7.3, 7.5) is encapsulated within the payload.

4.4 Administrative response block

Fields are replicated from the originating command block (4.3). Function identification, domain, module identifier, transaction counter are all reported by the module current state, even if they were missing from the request. The response is augmented with the PKCS#11 return code; payload is the field returned by the query or command.

```
xcpAdminRspBlk ::= SEQUENCE {
    admFunctionId     OCTET STRING,   -- command/query identifier
    domain            OCTET STRING,   -- administrative domain (6.2.1)
    moduleIdentifier  OCTET STRING,
    transactionCtr    OCTET STRING,   -- current/updated value
    response          OCTET STRING,   -- return value (CKR_...)
    payload           OCTET STRING    -- command-specific interpretation
}
```

Domain-level responses fill in the entire domain field, including instance identifier.

As with requests, the interpretation of ‘‘payload’’ is query/command-specific.

4.5. Administrative structures

4.5.1. Integer administrative attributes

Individual attributes are encoded as a fixed-size 32-bit raw integer (index), followed by the 32-bit raw integer value, without further encapsulation. Multiple attributes are encoded as a packed array, concatenating individual attributes without padding or other formatting.

When changing attributes, only the updated ones need to be present. Setting attributes must not repeat an attribute, even with identical value (i.e., each attribute must appear at most once in the payload). Queries return all available attributes.

A listing of supported attributes is under (8.1.1). As with other similar fields, this list may grow in the future, but assigned values will retain their meaning.

4.5.2. Control points (CPs)

When a full set of control points is serialized, they are encoded as a fixed-size bit array, an integer multiple of 128 (XCP_CPBLOCK_BITS) bits. The actual number of supported CP bits is supplied through a module-level configuration query (5.1.1); the backend requires a full set to have the proper padded size.

The field is filled from the left, bit-wise, i.e., a field with only CP number 1 active contains "40 00 00..." hex (CP numbering is zero-based).

The set must contain zeroes for CP bits which are currently not set. The backend always returns such CP sets. Active CP bits which the backend does not recognize are ignored: they may be set, but read back as all-zeroes. This way, backends will tolerate CP sets from the future. (Note that the number of supported CPs may be queried.)

If CPs which must not be set are provided in a set request, they are ignored without an error. Host code will be find these bits to be missing in a subsequent CP query. (Rationale: supported, never-settable CP bits are handled the same way as CPs from the future are: ignored, not set.) Such automatically removed CPs may be driven by card policy, including minimum compliance levels, capability-controlling ‘‘Function Control Vectors’’ (6.13.3), or other module-internal policy enforcement.

See XCP_CPB_... definitions for individual CPs (8.4)

5. Compound structures

5.1. Query types

The following structures are reported by the available queries, which are selected as parameters to get_xcp_info() (6.13). Note that some of the categories are optional, their presence reported through extended token flags (6.13.1).

5.1.1. Module query

The module information query returns the following fields, concatenated in this order without any padding or further formatting:

	offset	field	bytes	notes
---	backend revision/configuration			
1	0	API ordinal number	4	only LS 16 bits are nonzero
2	4	firmware identifier	4	truncated FWID (6.14)
3	8	API version, major	1	
4	9	API version, minor	1	
5	10	CSP version, major	1	CSP-specific meaning
6	11	CSP version, minor	1	
7	12	firmware configuration	32	FWID, combines xcp and CSP (6.14)
8	44	xcp configuration	32	hash, xcp code without CSP
9	76	CSP configuration	32	hash, included CSP library
---	device identification			
10	108	serial number	16	device || instance (6.3)
11	124	module date/time	16	UTC (6.11)
12	140	operational mode	8	standards’ compliance (8.1.1.3) aggregate of domains’ settings
---	infrastructure properties			
13	148	PKCS#11 flags	4	
14	152	extended flags	4	non-standard capabilities (6.13.1)
15	156	domain count	4	number of supported domains
16	160	incremental symmetric state	4	symmetric incremental en/decrypt
		bytecount	4	and sign/verify (such as CMAC)
17	164	digest/HMAC state bytecount	4	
18	168	pin blob bytecount	4	
19	172	SPKI bytecount	4	incl. salt, attributes, and MAC

20	176	private key blob bytecount	4	
21	180	symmetric blob	4	raw symmetric keys (any type)
22	184	payload size limit	4	maximum gross payload bytecount which may be passed to channel (payload only, without channel headers); 0 if no backend limit
23	188	CP profile bytecount	4	XCP_CPID_BYTES of the backend 0 if profiles are not supported
24	192	maximum CP index	4	
---	end			
	196	(total bytecount)		

XCP and CSP configuration is reported separately, in addition to the compound FWID, to allow tracking of different configurations of the same code, such as HSM-backed and soft-HSM instances built from the same XCP revision. The compound FWID uniquely identifies the entire backend.

CP fields (4.5.2) are padded to units of 128 (XCP_CPBLOCK_BITS). The structure returns the net CP count, the number of control points actually available (note: not the index of the last one).

Bytecounts are reported as possible maximum values for the given type.

The corresponding PKCS#11-similar structure in ep11.h is CK_IBM_XCP_INFO. It is loosely based on the CK_INFO structure returned by the PKCS#11 C_GetInfo call (which XCP does not support).

5.1.1.1. Text/description fields

Inherited from PKCS#11, a number of fields targeted for human/log consumption may be queried as a sub-query (9 (CK_IBM_XCPQ_DESCRTEXT)).

The backend does not interpret any of these fields. The are not intended for machine interpretation, other than depositing them into host logs. We do not specify any particular subdivision of the fields.

	offset	field	bytes	notes
1	0	manufacturer ID	32	see PKCS#11 token/slot manufacturer
2	32	model identifier	16	see the ‘‘model’’ field of PKCS#11
3	48	token/slot label	32	
	80	(total bytecount)		

Fields SHOULD contain only valid UTF-8 encoded data; this expectation matches that of PKCS#11, but it not enforced by the backend.

5.1.2. Domains query

This structure, reported by the CK_IBM_XCPQ_DOMAINS query type, returns a packed array of information fields about domains. Its purpose is to provide an overview of domains. The response contains zero or more of the following records, concatenated without further formatting or padding:

	offset		bytes	note	
1.	0	domain index	4	raw domain	(6.2)
2.	4	domain WK ID	4	32 MSB of WK ID	(6.7.1)

Domains without administrators or any of the WKs are skipped. The returned list is in increasing index order.

5.1.3. Domain query

Domain queries return the following properties of the targeted domain, concatenated in the following order, without further formatting or padding:

```
    offset                          bytes  note
    --------------------------------------------------------------------------
1.  0      domain index             4      raw domain
2.  4      current WK VP            32     WK hash, active WK     (6.7)
3.  36     pending WK VP            32     WK hash, pending WK    (6.7)
4.  68     domain flags             4      admin/WK presence      (6.13.2)
5.  72     operational mode         8      standards' compliance  (8.1.1.3)
    --------------------------------------------------------------------------
    80     total bytecount
```

Multi-byte fields are big-endian.

The domain is redundant in the reply. It is replicated to make the response selfcontained, meaningful without the entire encapsulated response.

Key verification patterns are zero-filled if the corresponding WK is not present (see the domain flags field).

Note that this query result is not signed. If a trustworthy, signed copy is needed for audit purposes, one should issue the corresponding administrative query.

5.1.4. Selftest

Selftests do not return any results, only CKR_OK if tests pass, or CKR_FUNCTION_FAILED if any selftest or known-answer test fails. The latter return value is used for other purposes by other commands, but those conditions may not be reached by a query.

In production builds, we do not provide further details about which component failed in the response code, but we may do in other logging channel such as system log.

5.1.5. Audit records history

The backend retains the last audit records, and makes them available through a query type, returning one record per query. The sub-query (type) selects the record, with a 1-based index starting at the most recent record. Sub-query type 0 returns the number of currently stored records as a 4-byte raw integer.

If the the requested index is out of range, CKR_KEY_HANDLE_INVALID is returned (note that the indexed object is, obviously, not a key).

This query type is conditional, it is rejected by backends which do not store audit chains.

5.1.6. Host queries

A range of query indexes are reserved for host extensions, which should not be passed down to the backend, and are not available there. Query indexes of 0xff000000 or higher (CK_IBM_XCP_HOSTQ_IDX) are reserved for this range, and they are expected to be intercepted by host. Backends will never react to queries within this range.

Host queries are beyond the scope of this document, but we maintain some recommendations for host queries, and would prefer to include host-originated information in our future releases.

As a minimum, host libraries SHOULD include a query to enumerate the number of supported host queries supported. When further constants are supported, they SHOULD match the reserved constants enumerated under (8.7.1.2). Our host code returns any host-query response as a raw buffer, without any interpretation.

5.1.7. Performance classification statistics

The backend aggregates performance statistics, keeping a count of requests which belonged to each defined performance category (8.2.1.). the query returns a packed array of 5 (XCP_OPCAT_ASYMM_MAX) 32-bit entries, without further formatting or padding, in increasing category order.

Note that individual 32-bit counters wrap without any notification. In practice, we expect short-term measurements to be unaffected, if they identify counter wraps (and MAY assume each counter-wrap event means an increment of 2^32 events).

When high-throughput backends will approach realistic limits of 32-bit counters, counters will be extended to 64 bits. A new query will be added then, returning 64-bit counters, and this base query will return the least significant 32 bits of each counter.

Since queries are interleaved with functional requests, the host MUST assume values are approximate (in other words, expect exact results only from an otherwise quiesced module).

5.2. Development test structures & calls

Development backends support a test function which allows potentially dangerous test operations. All such test operations take an input field, at least 4 bytes, with the first 4 bytes containing a 32-bit raw integer selecting the operation. Supported operations are enumerated within XCP_DEVcmd_t values (8.6).

Development functions are passed through the non-standard EncryptSingle() or DecryptSingle() calls, using CKM_IBM_TESTCODE (0x8001000e) as a custom mechanism (see also 8.7.3). When using this test mechanism, packed test structure must be passed as plaintext; the blob parameter is ignored by most calls---see iterated en/decryption (5.2.4) for the single counterexample.

Note that the actual configuration of supported development extensions may be setup-dependent. Please check with your system administrator if an expected service is not supported. We do not currently support a capabilities query for development extensions, but it may be added in the future.

Except for test functions described below, test calls do not take input parameters or return data.

5.2.1. Set current WK (both forms)

This format includes both raw and imprinting set-WK variants. (They both set a domain WK, but one also moves the domain out of imprint mode.)

Raw bytes of the current WK follow the development operation ID. The following bytes are raw bytes (32), without further formatting.

The returned result concatenates the newly loaded WK, and the internal MAC key derived from it.

5.2.2. Set pending WK

Raw bytes of the pending WK follow the development operation ID. Note that this payload is only accepted if there is a current WK in the targeted domain.

As with current WKs, raw keybytes follow the operation ID, without further formatting.

The returned result concatenates the pending-WK which has been accepted, and the internal MAC key derived from it.

5.2.3. Set control points (CPs)

The full set of newly active CPs follows the development operation ID (4.5.2.). The supplied CPs replace the ones within the targeted domain.

The full CP set must be present; it is verified as documented under the corresponding administrative command.

The returned result is the CP set which has been loaded.

5.2.4. Iterated encryption/decryption

These cryptographic calls perform repeated operations on the same data; their primary use is for performance measurements and compliance testing, such as power analysis. This is the only test call which uses the blob parameter, containing the key being tested; its type must be consistent with the test function identifier (AES or T/DES).

Input data following the function identifier is interpreted as a 32-bit raw integer (first 4 bytes) specifying iteration count, followed by the single block of input (plaintext or ciphertext). Block size is determined by the cipher/key type.

5.2.5. Manage read-only mode

This test call manages internal filesystem access, selectively restricting access to filesystems. It may be used to simulate certain stages of a concurrent-update process, where internal filesystems may not be writable, or may be completely unreachable.

The call uses a single, 4-byte raw integer as parameter, both as input, and output. The following values force restrictions as follows:

XCP_DEVFS_READONLY (1) -- writes prohibited
Calls restoring data from filesystems are allowed.

XCP_DEVFS_NOACCESS (2) -- no filesystem access
All persistent-filesystem reads and writes are prohibited.

In both restricted modes, memory structure updates are allowed.

The XCP_DEVFS_QUERY constant (0) only reports
current state, without updating it.

All forms return the current setting, as a single, 4-byte raw integer.

5.2.6. Blob and CSP configuration query

This query returns information about the internal configuration of blob
data, including CSP objects. While these details are kept opaque for
production builds, the test query reports them for tool use---such as
our system fuzz-tester.

	offset		bytes	note
1.	0	structure version	4	currently, fixed v1
2.	4	flags	4	see (5.2.6.1)
3.	8	object marker bytecount	4	header-internal marker
4.	12	integer bytecount	4	
5.	16	size_t bytecount	4	
6.	20	void pointer bytecount	4	
7.	24	function pointer bytecount	4	
8.	28	mutex pointer bytecount	4	0 if backend does not support locks
9.	32	object tail bytecount	4	marker follows object, if non-0
10.	36	header total bytecount	4	including any padding
	40	total bytecount		XCP_CSP_CONFIG_BYTES, see (8.7.)

Multi-byte fields are big-endian.

5.2.6.1. Blob configuration flags

The following additional flags are defined for the blob configuration query:

1 if backend is little-endian

5.2.6.2. Blob object header configuration

CSP object headers concatenate a marker (if bytecount is >0), two
integers, a size_t net object bytecount, a pointer (context), a function
pointer, and a mutex pointer (if bytecount is >0), in this order.
Padding rules of the backend apply.

Note that some fields within the blob-internal header are ignored,
since they are filled during blob decryption.

5.2.7. Algorithm tests

The test interface provides direct access to some algorithmic
primitives, such as needed for algorithm testing.

5.2.7.1. RSA key generation (ANSI x9.31)

ANSI x9.31 key generation enumerates prime candidates based on prime
factors p,q and a random X0, which has the bitcount of the targeted

prime.

```
    offset                          bytes  note
    ------  header  ---------------------------------------------------------
1.       0   structure version          4  currently, fixed v1
2.       4   prime-factor bitcount      4  of Xp1, Xp2, Xq1, Xq2
3.       8   prime bitcount             4  of Xp, Xq
4.      12   public exponent bitcount   4  possibly 0 (if using Fermat4)
                                           up to 256, inclusive
    ------  prime-search parameters  ----------------------------------------
5.      16   prime factor base Xp1   <var>  search base for prime factor p1
6.  <var>    prime factor base Xp2   <var>  ...p2...
7.  <var>    prime base        Xp    <var>  search base for prime P
8.  <var>    prime factor base Xq1   <var>  ...q1...
9.  <var>    prime factor base Xq2   <var>  ...q2...
10. <var>    prime base        Xq    <var>  search base for prime Q
11. <var>    public exponent   E     <var>  possibly missing: default Fermat4
                                           (0x10001) is used if not specified
    --------------------------------------------------------------------------
```

Parameters are concatenated without further padding or formatting. Prime factors and prime bases must be zero-extended to uniform length of their respective category. In practice, these parameters are generated with fixed bitcounts, and zero-padding bytes would be seldom needed.

Multi-byte integers are big-endian.

The returned private key is encoded in PKCS8 format, in the clear. Remember that this is a test-only operation, not shipped with production builds.

Note that the generated primes P and Q may be reordered, to force P>Q, if the CSP requires this (it is standard practice). In such cases, swapping the sections for P and Q will result in identical PKCS8 output.

5.2.7.2. DSA parameter (PQG) generation

DSA PQG generation may be called with a seed, iteration count, and P+Q bitcounts. It returns the generated PQG tuple, with the trailing iteration count, or an error if the search for PQG from the given seed and iteration count does not terminate.

```
    offset                          bytes  note
    ------  header  ---------------------------------------------------------
1.       0   structure version          4  currently, fixed v1
2.       4   prime bitcount             4  P
3.       8   sub-prime bitcount         4  Q
4.      12   iteration count            4  number of Q candidates to test
    ------  seed  -----------------------------------------------------------
5.      16   initial seed           <var>  all remaining bytes
    --------------------------------------------------------------------------
```

Multi-byte fields are big-endian.

The seed field may be missing/empty. All bytes following the header are passed to PQG generation as seed (note: this field is hashed, so there are no practical limits on its length).

If the prime, sub-prime bitcount is unsupported, the request is rejected with CKR_ATTRIBUTE_VALUE_INVALID.

```
5.2.7.3.  DSA PQG parameters, response

Upon successful termination, PQG parameters are returned with the
counter value, and the final seed:

xcpReq ::= SEQUENCE {
    pqgParams  OCTET STRING, -- contains DSA PQG, 1.2.840.10040.4.1 ASN/BER SEQ
    seed       OCTET STRING,
    counter    OCTET STRING
}

Note that these response parameters correspond to NIST PQG-generation
test-response file fields.

5.2.7.4.  EC scalar multiply, looped

This is a test function, added for side channel analysis. It must
be paired with an initialized EC private key, and calculates k*Q
(user-provided curve point) or k*P repeatedly.

    offset                        bytes  note
    ------  scalar  ----------------------------------------------------------
1.       0  structure version         4  version: 1 if calculating k*P,
                                         2 if k*Q (user-provided point)
2.       4  scalar                <var>  byte count as appropriate: size of
                                         base prime, zero-extended if needed
    ------  point, if present  -----------------------------------------------
3.  <var>   X coordinate          <var>  present only in v2; byte count is of
                                         base prime; zero-extend to full length
4.  <var>   Y coordinate          <var>  see X
    --------------------------------------------------------------------------

The return value is a concatenated X || Y coordinate pair, both components
zero-extended to prime length.

5.2.8.  Performance-test events

Performance-test events form a circular buffer, allowing query access to
the last events. As input, performance-test queries require two 32-bit
raw integers, requesting start offset (0 meaning the most recent entry)
and the number of events requested.

Similar to file-part queries (6.17), responses use a fixed header,
optionally followed with data.

    offset  field                      bytes  note
    ------  header  ---------------------------------------------------------
1.       0  performance-test ID            1  ASCII 'P' (0x50) or 'p' (0x70)
2.       1  performance-test status        1  ASCII '0', 'a', or 'b', see below
3.       2  measurement event type         1  ASCII '0', 't', or 'T', see below
4.       3  reserved                       1  ASCII '0' (0x30)
    ------  event count  ----------------------------------------------------
5.       4  number of events               4  raw integer
6.       8  offset of first event          4  raw integer
    -------------------------------------------------------------------------

When passed an empty request, or two 32-bit zeroes, the response
consists of an empty header. Otherwise, if the requested range is valid,
measurements follow the header (5.2.8.1).
```

Response header concatenates a fixed identifier, a single byte indicating test status, a byte indicating type of events, and a reserved byte, in this order, in the first four bytes.

Performance-test responses use the ASCII identifier 'p' (0x70) when reporting only capabilities, and ASCII 'P' (0x50) when returning actual measurement events. No data follows the header in the first case. An integer number of measurements (5.2.8.1) follow as a packed array in chronological order.

Test status is ASCII '0' (0x30) if the backend does not support performance measurements, ASCII 'a' (0x41) if measurements are available and measurements are currently being collected. Status changes to ASCII 'b' (0x42) when measurements are available, but collection is currently switched off. This is recommended when measuring performance on an active system, to remove interference from unrelated requests.

Measurement event type is ASCII '0' (0x30) if the backend does not support performance measurements, ASCII 't' (0x74) if timestamps are returned without high-resolution cycle counters, and ASCII 'T' (0x54) if measurements include the cycle counter.

Note that performance tests are currently a development-only feature, and are used only in controlled environments, therefore we currently do not provide details of the backend (such as cycle counter frequency etc.). Additional information may be later provided if this feature becomes available in production builds.

5.2.8.1. Performance-test measurements

Each measurement is encoded as 16 bytes, concatenating a 32-bit event identifier, a 64-bit value encoding a high-resolution UTC clock reading (6.11.2), and the least significant 32 bits of high-resolution timestamp. Fields are concatenated without further formatting or padding:

	offset	field	bytes	note
1.	0	event identifier	4	no further specification; recommended to be hierarchically assigned by development
2.	4	time_t, seconds	4	parsing must handle rollover field is not Y2038-safe
3.	8	time_t, sub-seconds	4	
4.	12	high-resolution timestamp	4	least significant 4 bytes only

When performance measurement is in fast mode, time_t fields are left all-zeroes. There is no additional indication of this happening; in fast mode, host code most reconstruct the entire wall-clock concext, relying on cycle counters only.

High-resolution fields are zero-filled if the backend lacks a high-resolution counter. Backend platforms with cycle counters wider than 32 bits store only the least significant 32 bits.

5.2.8.2. Host-influenced delay (NOP)

The test operation ''XCP_DEV_DELAY'' introduces a host-influenced amount of sleep-wait to the backend; this feature is used in stress-testing and threading diagnostics. The host may select the probability of delaying execution, and supply lower+upper bounds :

	offset	field	bytes	note
1.	0	delay probability	4	PPM, probability 'N in a million'
2.	4	minimum delay, microseconds	4	
3.	8	maximum delay, microseconds	4	

When encountering invalid fields, the backend returns without delaying.

Actual delay is subject to backend-timer resolution.

5.2.9. Performance of raw primitives

Development-only tests for locking/thread-wakeup etc., and related primitives are available. These performance-measurement tests all take a single parameter, iteration count, as a 4-byte raw integer.

The current list of supported primitives is:

```
    XCP_DEVQ_PERF_LOCK
    XCP_DEVQ_PERF_WAKE
    XCP_DEVQ_PERF_SCALE  (accepts but ignores iteration count)
```

Please check source and header files of test code for implementation details (these functions are not available in production, and lack public documentation).

5.2.10. Blob-cache configuration

The development-only services interacting with module-internal blob (cleartext) caching use a bitmask to set/indicate the state of caching.

A single service, XCP_DEV_CACHE_MODE (0x00000021)
doubles as get/set call, depending on the 4-byte parameter supplied to it. When supplied an all-zero value, it is only a query; otherwise, it interprets the supplied 32-bit raw integer as a bitmask (8.6.2). In both forms, the response contains a 32-bit bitmask, containing exactly one of 'active' and 'suspended' bits. If the supplied bitmask contains unknown bits, or contradictory ones, results are undefined.

The related statistics query, read through XCP_DEVQ_CACHE_STATS (0x00000022), prompts the backend to log its then-current cache statistics over any logging facility available to it. In a typical HSM instance, this would mean use of syslog entries through the host system.

Note that in production builds, blob caching is kept transparent to the host, therefore there are no production equivalents of these services.

5.2.11. Set/erase/query FCV

When calling the test function XCP_DEV_FCV (0x0000002b), the
caller MAY include an FCV in the call parameter, or supply no additional data to restrict the service to a query.

When supplying a new FCV, we expect the full, signed structure
(6.13.3.), BUT DO NOT VERIFY ITS SIGNATURE. This method still validates fields of the newly loaded FCV, so it may be used to test FCV-checking code, even if it bypasses signature verification. This mode performs all

size and sanity checks on the supplied input, so there are no predefined wire-visible restrictions on payload size or structure (all checking follows the FCV specification).

When no data is supplied, other than the function ID, the set-FCV service reduces to a query. This is the non-administrative equivalent of an FCV query; other than the missing administrative signature, it is functionally identical to the administrative query (see XCP_ADMQ_FCV).

The erase-FCV service takes no input data; it removes any currently loaded FCV.

All FCV-related test functions respond with the currently loaded FCV (6.13.3.), or no data if the FCV has not yet been initialized. In addition to infrastructure errors, set-FCV may return values caused by FCV (structure) validation.

Note that during regular operations, only the first FCV is accepted, further changes are rejected (submitting the same FCV multiple times is tolerated). The capability to change FCVs at runtime is unique to test builds: it has been added to exercise the FCV-processing state machine without major infrastructure changes.

5.3. Serialized module state

Module state, when serialized, consists of a sequence of TLV (ASN.1/BER) encoded sections, each an ASN.1 OCTET STRING, encapsulated within a single SEQUENCE. The Value field contains type-prefixed sections, starting with a two-byte type prefix, a 32-bit raw integer identifier, which are optionally followed by data. Size and presence/lack of the data field is inferred from the TLV structure (i.e., the length of the OCTET STRING).

Type identifiers imply versioning; if fields/data formats are later extended, they will be introduced under different types, and migration rules will be defined for them. Certain types will only appear once in an exported state; others, such as domain-specialized instances of the same type, may appear multiple times---with the integer identifier designating the specific instance.

The combination of types and identifiers (i.e., leading 2+4 bytes) is unique, by construction. As a consequence, sections may be processed in arbitrary order.

Note that (parts of) certain sections are redundant. Since state is transported in pieces, and there are no restricting storage limits---practical sizes are moderate---we provide redundant fields to allow easier parsing, and tolerate the additional bytes As an example, creation date is assigned with its own section x0005, allowing readers to skip parsing another section with a structure including module time (x0007).

Supported sections are (see (8.1.2.) for a list of symbolic names):

type	may	repeat?	
(hex)		integer ID	data/notes
x0001	n	total number of sections (including this one)	most significant 32 bits of SHA256 hash of all following bytes, but excluding any file signatures

```
                                 always the first section (if present)

x0002  n  largest domain index    total number of included domains
                                  (32-bit raw integer)
x0003..n..smallest domain index...bitmask of domains present in serialized
                                  state; bit 0 corresponds to smallest
                                  domain index (cf. offset and bitmask
                                  fields of a domain mask window, see 6.15.2)

x0004  n  reserved 0              serial number of originating module
                                  excluding instance identifier (see 6.3)
x0005  n  reserved 0              state creation date/time (6.11)
x0006  n  reserved 0              FCV structure of originating module,
                                  public parts only (same as query FCV
                                  structure, see 6.13.3)
x0018..n..4-byte salt.............(no additional data)
                                  random value generated by exporting
                                  module, potentially used as export-unique
                                  identifier

x0007  n  reserved 0              card query structure (5.1.1)
x0008  n  reserved 0              card administrator SKIs, packed
x0009..n..reserved 0..............card administrator certificates, packed
                                  same order as SKIs under type x0008

x000a  Y  domain                  domain administrator SKIs, concatenated
x000b  Y  domain                  domain administrator certificates,
                                  concatenated.  Entries are in the same
                                  order as SKIs under type x000a
x000c  Y  domain                  domain query structure (5.1.3), excluding
                                  leading domain index. Together with domain
                                  within integer ID, structure is the exact
                                  domain query structure.
x0017..Y..domain..................control points

x000d  n  threshold               SKIs of targeted KPHs, concatenated
                                  (in order of index);
                                  threshold is number of KPHs needed
                                  to reassemble full key
x000e  n  reserved 0              card attributes, full listing (4.5.1)
x000f..Y..domain..................domain attributes, full listing (4.5.1)

x0010  n  reserved 0              card transaction counter
x0011..Y..domain..................domain transaction counter

x0012  n  WK encryption alg ID    file IV: initialization vector (or other,
                                  algorithm-dependent initialization data)
                                  size of data is algorithm-dependent
x0013  n  reserved 0              encrypted WKs and next-WKs.  See type
                                  x0012 for algorithm and parameters (i.e., IV).
x0014  n  cert chain elements     (no data); integer ID is the number of
                                  certificates in signing certificate
                                  chain (i.e., OA, or other backend
                                  attestation mechanism)
x0015..Y..cert chain index (N)....backend certificate number N (see type
                                  x0014 for maximum index).  Certificates are
                                  indexed from current key to
                                  the root (see 6.12)

x0019  Y  index                   encrypted keypart (RecipientInfo);
```

```
                                  indexes are unique
x001a  Y  index                   signature on encrypted keypart
                                  index corresponds to that of related keypart
x001d  Y  index                   certificate of the targeted KPH (keypart
                                  holder), with host-issued index.  See
                                  (5.3.4) for details of KPH certificates.
x001b..N..count...................(no additional data)
                                  total number of keyparts
x001c  N  count                   (no additional data)
                                  number of keyparts needed to reconstruct
                                  encryption key

x001e  N  reserved 0              raw certificate of CA (certificate authority)
                                  issuing some of the certificates used by
                                  key transport.  This host-supplied data
                                  is included verbatim; it is not used
                                  by state-migrating code itself.

x001f  n  32-bit bitmask          restrictions on the scope of sections
                                  included in serialized state, if applicable,
                                  see (5.3.5) for interpretation.
                                  Included only if the originating export
                                  request included restrictions.

x0021  n  number of bits set      bitmask of all CPs supported
   in bitmask (be32)       by the originating module (8.4,
                                  see (6.18) for encoding)
                                  see the XCP_STSTYPE_CARD_QUERY section type
                                  (x0007) for
                                  the structure reporting CPB count

----- --- ---------------------- --- (signature) ----------------------------
x0016  n  reserved 0              signature of all preceding bytes, signed
                                  by originating certificate (see type x0015)
                                  always the last section (if present)
```

Except for the signature section (type x0016), and the first section containing the truncated file hash (type x0001), relative ordering of sections is undefined.

The file checksum, if present, covers between the first byte of the TLV-encoded section immediately following the checksum (i.e., the first tag byte of the second section) and the last byte of the TLV-encoded section preceding the signature. It serves as a quick integrity check, and an immediately visible, data-dependent identifier, but serves no cryptographic purpose. The value may safely be ignored; file integrity is obviously properly covered by the digital signature.

If the FCV setup of the targeted module is more restrictive than that of the originating module, import is rejected.

5.3.1. Tags within module state sub-types

While sharing encoding, different subsets of possible tags are used in different stages of state transport. This section enumerates types possibly included in each kind of data. Other types may also be present, and are ignored.

Some of the types are partially redundant (such as KPH certificates and SKIs, at different states of the migration process). Unless specified otherwise, redundant structures are not checked for consistency (in such cases, we mention the type which will be interpreted, all others are ignored). Note that such redundancy is restricted to the exported state itself, which intentionally contains redundant fields to ease processing, and much of the included metadata is ignored by the importing module.

5.3.1.1. Request to export

When requesting a state export, the following types must or may be present in the export-request collection:

```
---  mandatory  -------------------------------------------
1  XCP_STSTYPE_KEYPART_CERT   x001d  certificates of KPHs who will receive
                                     one encrypted KP each
2  XCP_STSTYPE_KEYPART_LIMIT  x001c  number of KPHs required to reassemble
                                     full transport key
---  optional  --------------------------------------------
3  XCP_STSTYPE_DOMAINS_MASK   x0003  domain bitmask (6.15)
4  XCP_STSTYPE_CERT_AUTH      x001e  certificate authority cert, if included;
                                     note that the certificate is not used by
                                     EP11 itself
5  XCP_STSTYPE_KEYPART_COUNT  x001b  field is redundant, but tolerated
                                     and verified if present, see below
6  XCP_STSTYPE_STATE_SCOPE    x001f  scope restrictions of the originating
                                     export-request, if applicable (5.3.5)
-----------------------------------------------------------
```

The KPH certificates included in the request must be supplied with different indexes between 0 and N-1, inclusive. The response structure will preserve the same indexes for keypart-related types. (Note that this index may differ from the index---the x coordinate---used by the secret-sharing process.)

The number of KPs needed to reconstruct the full transport key must be in the range 1 to N, inclusive.

If no domain mask is present, the source module exports data from all domains which contain administrator certificates, or have non-default domain-attribute setups. (Any of these two indicate the domain has been used or populated.)

If a domain mask has been specified, the exported set of domains is used as-is, without further filtering.

The CA certificate, if included, will be included in serialized state. Contents are reused verbatim, without any modification. This field is used where procedural migration controls are tied to verification of certificate chains from specific roots (‘‘MCA certificate’’, in the original terminology used by state migration). Our backend code does not use or verify any signatures from this certificate.

The total count of KPH certificates may be included in the request, and if present, it must the number of certicate sections. This redundancy is allowed for documentation purposes, if the requestors wish to ‘‘document’’ the number of certificates in some obvious way.

Other sections, if encountered within the request file, are ignored.

Note that non-sensitive state export MUST NOT include keyparts or

```
encryption, and therefore mandates a much smaller set of sections
(5.3.1.2).

5.3.1.2.  Request to export non-sensitive state

Since non-sensitive state export does not involve encryption and
therefore keypart-holder identification, only the following fields
are required:

---  optional  -------------------------------------------
1  XCP_STSTYPE_DOMAINS_MASK   x0003  domain bitmask (6.15)
---  de facto mandatory  ----------------------------------
2  XCP_STSTYPE_STATE_SCOPE    x001f  scope restrictions of the originating
                                     export-request, if applicable (5.3.5)
------------------------------------------------------------

While the section restricting scope is nominally an optional one (5.3),
the default scope is without restrictions. Therefore, in order to
trigger a non-sensitive state, the export request must have included a
scope-restricting section.

5.3.1.3.  Request to import into multi-domain setup

When requesting import of a single-domain exported state into specific
domain/s, the following section must be included in the administrative
structure requesting import:

------------------------------------------------------------
1  XCP_STSTYPE_MULTIIMPORT_MASK  x0020
                                  Supported only in state-import requests

                                  Data field must contain a full,
                                  valid domain mask (6.15).

                                  Presence of this section designates state
                                  import request as a single-domain source,
                                  replicating all domain input fields from the
                                  imported state to all enumerated domains.
                                  Module-level sections are ignored during
                                  import. All domain-level sections present
                                  must be from the same source domain (i.e.,
                                  we replicate data from a single exported
                                  domain to one or more domains,
                                  specified through mask)
------------------------------------------------------------

This section is singular, not domain-specific. The integer ID field is
ignored during import; administrators may safely use it to store any
related 32-bit data, such as the number of domains targeted.

5.3.1.4.  Serialized state

State data is signed by the originating module, and must be passed to
import without modification. Therefore, type composition during export
is obviously identical to the import version.

Certain fields of the exported state---such as creation time and
salt---are ignored by the importing module.

5.3.1.5.  Exported keyparts
```

```
As with other typed formats, the order of sections within exported
sections is mainly arbitrary, with the exceptions listed in (5.3).

---  always present  ------------------
1  XCP_STSTYPE_SECTIONCOUNT   x0001  file header, incl. number of tags
2  XCP_STSTYPE_KEYPART        x0019  repeated for each KPH recipient
3  XCP_STSTYPE_KEYPART_SIG    x001a  card (OA) signature on corresponding
                                     KEYPART section (full encrypted field)
4  XCP_STSTYPE_KEYPART_CERT   x001d  certificates of KPHs who will receive
                                     one encrypted KP each
---  for audit purposes  -------------
5  XCP_STSTYPE_KEYPART_LIMIT  x001c  number of KPHs required to reassemble
                                     full transport key
6  XCP_STSTYPE_KEYPART_COUNT  x001b  number of KPHs (redundant)
7  XCP_STSTYPE_CREATE_TIME    x0005  file timestamp
8  XCP_STSTYPE_STATE_SALT     x0018  package salt
9  XCP_STSTYPE_SIG_CERT_COUNT x0014  source module cert chain size
10 XCP_STSTYPE_SIG_CERTS      x0015  source module certificates
11 XCP_STSTYPE_KPH_SKIS       x000d  (redundant, see below)
---  optional  -----------------------
12 XCP_STSTYPE_CERT_AUTH      x001e  certificate authority cert, if was
                                     included within export request
13 XCP_STSTYPE_STATE_SCOPE    x001f  scope restrictions of the originating
                                     export-request, if applicable.
---  file signature  ------------------
14 XCP_STSTYPE_FILE_SIG       x0016  note: parts are also individually signed
---------------------------------------

Keyparts---and their signatures, in separate sections---inherit their
index from the export request, where each KPH certificate includes a
host-selected index (5.3.1.1), to simplify matching encrypted keyparts
to their recipients.

Note that the file contains redundant signatures: both individual
keyparts and the full compound are signed. Signatures are redundant
for procedural reasons: the file-level signature may be used for
auditing. For compatibility reasons, signatures on individual parts
are used by client---i.e., KPH---smartcards. With KPHs verifying
individual signatures, the file-level signature may be safely ignored
unless required for audit. Conversely, once the file signature has been
verified, one should assume individual KP signatures are valid (note
that KPHs do not rely on this, due to procedural limitations).

The certificate chain of the originating card is included for
auditability, and it matches the corresponding data in serialized state.
The field is replicated here to make the signature on the keypart-export
file publicly verifiable, and may be ignored by the host---especially if
certificate chains are cached (since then, these identical certificates
would already appear in the approved list).

Recipient KPH SKIs are actually redundant, included to simplify auditing
and fast public-key identification. Since this list will match the
list of KPHs encountered, in the same order---this is enforced by the
exporting module---it is safe to ignore this section, unless required
for audit purposes.

Creation time and file salt are included for completeness.  They simplify
tracking and auditing exported state and full keypart sets, and may be
safely ignored by host code (as mismatched keyparts and exported state
```

would be functionally detected).

Audit-only fields and signatures on the keyparts may be omitted entirely
in the import process.

5.3.1.6. Keyparts for import

The following types must be present in the file containing keyparts
for import:

```
---  mandatory  ----------------------
1  XCP_STSTYPE_KEYPART_CERT   x001d  certificates of originating KPHs,
                                     with host-issued indexes
2  XCP_STSTYPE_KEYPART        x0019  keyparts, with indexes corresponding
                                     to that of KPH certificates
---------------------------------------
```

The keypart collection for import must contain sufficient keyparts to
reassemble the transport key (see XCP_STSTYPE_KEYPART_LIMIT, type x001c).
The actual limit is present within the corresponding serialized state,
and therefore need not be included with keyparts.

Since keypart holders re-sign keyparts, most of the other metadata
present in exported keyparts is unnecessary for import (5.3.1.5).

5.3.2. File-content encryption

Currently, only one encrypted section type is defined, with a companion
section identifying algorithm, mode, and other metadata (currently,
the initialization vector used). If future formats add other encrypted
sections, we assume corresponding encryption-metadata sections will be
similarly added.

Formats append a MAC to ciphertext bytes without further formatting or
padding in an Encrypt-then-MAC construct. The MAC key is derived from
the encryption key.

Note that the MAC on encrypted state is orthogonal to the file
signature itself: it is tied to the transported unit, not the
originating card itself.

5.3.2.1. File-content encryption algorithms

The only currently defined algorithm is AES-256, CBC mode,
with unambiguous-random padding and HMAC/SHA-256 MAC, denoted by 0x00000001.

This algorithm implies a 16-byte IV, see state section x0012.

5.3.3. Lack of instance identifiers

Serialized system state excludes module and domain instance identifiers
of the source module. Since instance identifiers need to be queried
before authorized commands are submitted to the target module, full
state migration is possible while excluding them.

5.3.4. Keypart holder certificates

KPH certificates are used as public key stores, and no additional
information is verified about them. The certificates may be x509 SPKIs
of supported types (8.1.1.4) (7.4), or in the proprietary format used

```
by the TKE migration wizard (``migration certificates''). Existing
migration certificates are all from a subset of available importer
types.

Note that migration certificates include fields to specify number of
KPHs allowed and the possible threshold value. We do not directly use
this information, relying on state sections prescribing the limit (and
simply the number of KPH certificates setting the maximum number).

5.3.5.  State import/export scope restrictions

State export and import may be restricted in scope, if the section
indicating restrictions (x001f) is present.

If an export request contains non-empty, valid restrictions, the section
containing restrictions is replicated to exported state.

The following restrictions are defined:

XCP_STDATA_DOMAIN         0x00000001  -- Serialized state is restricted to
                                      -- domain data only, excluding module-
                                      -- specific sections. The resulting
                                      -- state structure may be safely
                                      -- imported without affecting module-
                                      -- global data, or unaffected domains
XCP_STDATA_NONSENSITIVE   0x00000002  -- Serialized state is restricted to
                                      -- non-sensitive sections only. Export
                                      -- structures so restricted may be used
                                      -- to quickly clone all administrative
                                      -- state to other modules/domains, without
                                      -- necessitating KPH involvement.
XCP_STWK_KP_NO_CERT       0x00000004  -- keypart export section is restricted
                                      -- to not return KPH certificates.
                                      -- (KPH certificates are already part
                                      -- of the request and are not necessarily
                                      -- needed within the response)

Bits are stored as the 32-bit subtype of the restriction section.
Multiple bits may be specified, enforcing the union of all applicable
restrictions. Export requests which include unknown restriction bits are
rejected. If an export request includes sections incompatible with the
requested restrictions, it is rejected.

If an import request encounters sections incompatible with restrictions,
it is rejected.


5.3.5.1.  Domain-restricted sections

Domain-restricted serialized state excludes the following sections:

XCP_STSTYPE_CARD_ADM_CERTS   x0009
XCP_STSTYPE_CARD_ADM_SKIS    x0008
XCP_STSTYPE_CARD_ATTRS       x000e
XCP_STSTYPE_CARD_TRANSCTR    x0010

Domain-restricted export requests do not prohibit any section.

5.3.5.2.  Non-sensitive restricted sections
```

```
Export requests for non-sensitive state MUST NOT include KPH
certificates, see (5.3.1.2.).

Serialized state restricted to non-sensitive data excludes the
following sections:

XCP_STSTYPE_WK_ENCR_ALG      x0012
XCP_STSTYPE_WK_ENCR_DATA     x0013
XCP_STSTYPE_KEYPART          x0019
XCP_STSTYPE_KEYPART_CERT.....x001d
XCP_STSTYPE_KEYPART_COUNT    x001b
XCP_STSTYPE_KEYPART_LIMIT    x001c
XCP_STSTYPE_KEYPART_SIG      x001a
XCP_STSTYPE_KPH_SKIS.........x000d

5.3.6.  Verification during state import

State import, since it replaces all administrative structures, may
override some of the restrictions.  Specifically:

1. transaction counters within the imported state are not compared to
   the current one: a restored state may roll the counters back compared
   to their then-current value (i.e., when the commit command is
   issued).

   Since transaction counters of the importing module, and those within
   the saved state are disjoint, this special case is not inconsistent.
   As instance identifiers are intentionally not transported within
   exported state, rolling back the transaction counters alone does
   not open up the newly installed module to replay attacks based on
   previously signed commands.

2. the importing module only verifies the full-file signature, but
   no signatures on keyparts, or any of the certificates. Since the
   import process itself is authenticated, all signatures within the
   imported state are effectively self-signed: the trust root of the
   originating module certificate chain may or may not be known to
   the importing one.  (OA trust roots, as an example, change between
   families, even if keysizes or algorithms are unchanged.)

   Since we are importing effectively self-signed data, we only
   verify the full-file signature as an integrity check.

Note that state-import is transactional: if a state-import procedure
would override the transaction counter, the original state is restored
after a failure. (Even if such failures are unexpected, the backend
covers the theoretical possibility.)

5.4.  Audit records

Audit records are generated for security-relevant events. The following
fixed-size structure is reported through system-logging facilities,
and may:

     offset
          field                        bytes  notes
----------------------------  header  ----------------------------------------
 1   0    record type                      1  hex x42 (ASCII 'b' from 'base log')
 2   1    record version                   1  0, currently
 3   2    record bytecount, total          2
 4   4    sequence number                6
 5  10    extended time_t                6  (6.11.1) (5.4.2)
----------------------  initial hash state  ----------------------------------
 6  16    initial state (hash)          32  (XCP_LOG_STATE_BYTES)
-------------------------  event context  ------------------------------------
 7  48    module identifier             16  including instance (6.3)
 8  64    audit instance                 2
 9  66    event type                     2  category, generic type (5.4.3)
10  68    firmware identifier            4  truncated 32-bit form (6.14)
11  72    event flags                    4  (5.4.5) (8.9.3)
12  76    function identifier            4
13  80    hosting domain                 4
--------------------  generic, flagged, optional fields  ---------------------
                                            For presence or absence of these
                                            fields: check flags (5.4.5)
                                            Order is fixed as shown here:

14  ..    WK identifier                 16  Means the original WK, if event
                                            describes a WK change
                                            (i.e., Finalize WK call)
15  ..    compliance                     8  ...of the hosting domain,
                                            if relevant
16  ..    final WK identifier           16  final WK, present only if changed
                                            during operation (i.e., Finalize)
                                            (XCP_WKID_BYTES)
17  ..    key record 0                  24  see (5.4.4)
18  ..    key 0 compliance               8  used if may differ from
                                            that of controlling domain
                                            (such as: key generation)
19  ..    key record 1                  24
20  ..    key record 2                  24
21  ..    final time                     6  used with time-changing
                                            administrative commands
22  ..    event details                  4  used when specific identified
                                            event caused audit entry (8.9.2)
23  ..    deterministic salt (PRF)       8  pseudo-random function (PRF) data
                                            derived from sequence number and
                                            timestamp fields (4, 5) is present
---------------------  end of non-salt, optional fields  ---------------------
24  ..    salt[0]                        4  see Audit record salting (5.4.6)
25  ..    salt[1]                        4
26  ..    salt[2]                        4
-----------------------  final hash state  ----------------------------------
27  ..    final state (hash)            32  (XCP_LOG_STATE_BYTES)
------------------------------------------------------------------------------
    290    maximal bytecount
```

The sequence number is a monotonously increasing counter, reset to zero during module zeroization. It is incremented by every issued audit entry. The counter wraps if it ever reaches its maximum value---which we believe to be infeasible. We consider the sequence number, internal time_t, and audit instance ID to be unambiguous, when processed together.

The audit instance is a salt, regenerated during each zeroization, or when the audit infrastructure itself is reset (which should not happen in the absence of file corruption). It works similarly to module instance identification (6.3), and has been added to separate audit-log lifecycles in a similar way.

Salt is inserted to prevent advancing the audit---hash---state with mainly host-controlled data (5.4.6). The number of actual salt bytes depends on event type; salt fields are inserted as appropriate.

Future fields will be added in chronological order, following any other optional fields, before---optional---salt. Applications may therefore interpret the recognized parts of audit entries if their exact size is known, by skipping unknown fields between optional ones and salt. The presence of unknown fields will be visible, when encountering unknown bits in the flags field.

The final state is a hash calculated over all preceding bytes. Within the module, it is saved as an updated state, and will be the initial state of the following audit event.

Multibyte fields are big-endian.

5.4.1. Audit record identification

The module instance identifier is only set after the audit infrastructure was started. Audit events that are issued before this point in time will have their module instance identifier set to zero.

Startup of the audit infrastructure can also encounter problems due to dependencies, such as failing to restoring module instance identifier. (Since module serial number and instance identifer originate in different sources, their restore is not atomic.)

5.4.2. Undefined audit record timestamps

If the module-internal clock query fails, audit entries are issued with all-zero time_t fields, but other fields are updated properly. Parsing code should accommodate these special audit records, which are logically invalid (time_t base is 1970-01-01, i.e., second 0 is in the past).

Since all audit entries are inserted into a hash-chain-based audit log, the lack of proper time does not endanger log integrity. Parsing code is encouraged to separately report these audit records, which will appear unusual also due to their non-monotonous clock sequence.

Note that failure of the internal clock is unexpected. However, this special case is documented for consistency.

5.4.3. Audit event types

The following events may be present in audit record. They describe the generic reason which prompted generation of the audit event.

XCP_LOGEV_QUERY	audit entry has been triggered by audit-specific query
XCP_LOGEV_FUNCTION	event for functional (non-administrative) function
XCP_LOGEV_ADMFUNCTION	event for administrative command
XCP_LOGEV_STARTUP	event related to module or subsystem startup
XCP_LOGEV_SHUTDOWN	event related to module or subsystem shutdown
XCP_LOGEV_SELFTEST	selftest results
XCP_LOGEV_IMPORT	import key, UnwrapKey() or administrative import
XCP_LOGEV_EXPORT	export key, WrapKey() or administrative export
XCP_LOGEV_FAILURE	error reporting
XCP_LOGEV_GENERATE	generate or derive new keys
XCP_LOGEV_REMOVE	erase an object
XCP_LOGEV_SPECIFIC	specific reason, see ‘‘event details’’

Constants are listed under (8.9.1)

5.4.4. Audit key-records

Key records identify keys within an audit entry. Key records are fixed-sized structures with the following structure:

```
     offset                         bytes
          field                        notes
------------------------------------------------------------------------------
 1   0    key type                  4  PKCS11 type (CKK_...)
 2   4    key size                  4  type-specific interpretation (5.4.4.1)
 3   8    controlling session       4  truncated, MS bytes of session
 4  12    key identifier            3  24 bits
 5  15    reserved                  1  reserved 0
 6  16    key MAC                   8  trailing 64 bits of object MAC
    24                                 total bytecount
```

Key type and size fields are big-endian.

Depending on the function, multiple key records may be present, such as key, KEK, and MAC key included in key un/wrapping audit entries.

As discussed under session identifiers (6.2.2), the session field is all-zero if and only if the object is not bound to a session.

Note that key-generation mechanisms are not included within key records. For audit entries where new keys are generated or derived, separate fields describe the generation mechanism.

5.4.4.1. Audit entries for key sizes

Audit records describe keysizes in type-specific forms.

Symmetric key sizes, including those of generic secret keys, are written as gross bitcounts. TDES keys' sizes include their in-band parity bits. For currently supported types, this notation corresponds to the CKA_VALUE_LEN attribute, expressing the size in bits instead of bytes.

RSA keys describe both their private and public exponent sizes: the size contains the public exponent (E) bitcount in its least significant 16, and modulus (N) bitcount in its most significant 16 bits.

DSA and DH keys describe both their private and public sizes: the size contains the modulus/base (P) bitcount in its most significant 16, and the prime/generator (Q/G) bitcount in its least significant 16 bits.

EC keys are identified through their base curve with a custom set of constants (8.1.1.5). Currently, custom curves are not supported.

5.4.5. Audit event flags

Flags are contained within a big-endian, 32-bit bitfield, which may contain combinations of the following bits (8.9.3):

```
WK is present               0x80000000
                             -- original/initial WK, if 'final WK present'
                             -- is also indicated (below)
compliance field            0x40000000
```

```
                              -- of the targeted domain
final WK is present          0x20000000
                              -- used only by events related to WK
                              -- transitions
Final time is present........0x01000000
                              -- time at completion is present
                              -- used by time-changing commands
Key record 0 is present      0x10000000
                              -- note that use of multiple key records
                              -- is specific to each key-using command
Key 0 compliance is present  0x08000000
                              -- key0 compliance bits are present
                              -- used only when they may differ from that
                              -- of the domain used (i.e., key generation)
Key record 1 is present      0x04000000
Key record 2 is present......0x02000000
Salt field 0 is present      0x00800000
Salt field 1 is present      0x00400000
Salt field 2 is present      0x00200000
Event details/reason present..0x00100000
Confounder/deterministic salt is present
                             0x00080000
                              -- deterministic salting is present
                              -- adds a representation of sequence nr+time,
                              -- with high apparent entropy (5.4.6.1)
```

Other bits within the field are reserved 0, and are not currently set by the backend.

5.4.6. Audit record salting

Audit records are constructed to include at least 96 random bits not determined by the host, to prevent record signatures from advancing the audit-hash state based on user-controlled data. Instead of arbitrary numbers of salt bytes, salt is added in 4-byte increments.

The relative order of salt fields is irrelevant. The backend currently fills them in increasing order; obviously, this distinction is no longer observable in the wire-encoded form [as these bits are generated by a cryptographic RNG].

5.4.6.1. Deterministic audit-record salt

An additional, optional field inserts deterministic, pseudorandom bytes into event records. Presence or absence of this field is currently determined by the backend, without host influence. The field insert a deterministic pattern which may possibly interfere with collision-constructing attacks, where one attempts to construct colliding event records leading to known [valid] final states.

When present, the field contains the Siphash-2-4 MAC of the wire-formatted ‘‘sequence number’’ and ‘‘extended time_t’’ fields, concatenated in this order, without further formatting or padding. [These bytes form a contiguoous region at fixed start offset and bytecount within event records.] We use an all-zero Siphash key to derive PRF bytes; the resulting bytes are encoded as big-endian.

Callers verifying event records SHOULD reconstruct and check the deterministic-salt field, when it is present.

The deterministic-salt field is NOT included in the salt bitcount (5.4.6). Since we treat the field as an extended version of the sequence number and timestamp, only in a representation with high apparent entropy, the actual function used to construct it is irrelevant, and it is excluded from compliance-related algorithmic restrictions.

5.4.7. Signed audit records

With the administrative query of audit history, the response payload encapsulates the queried audit record, the then-current backend time, and salt in an ASN.1/BER enclosure:

```
xcpReq ::= SEQUENCE {
    utcTime      OCTET STRING,     -- PKCS11 UTC format (6.11)
    salt         OCTET STRING,
    auditRecord  OCTET STRING...
}
```

Note that the time field follows regular PKCS11 formatting (6.11), not the compressed audit-specific form (6.11.1).

The response is salted to prevent signing all-known data. Salt size is fixed (12 bytes), but may increase in the future.

Audit records are inserted as the Value field of auditRecord, without further formatting or padding.

5.4.8. Audit record queries

When querying audit history, formatting of the query payload differentiate between an event-size query, or a request for a specific entry. Entries may be requested based on their relative index or their final state (hash).

Since element-count queries and subsequent entry queries are asynchronous, host code must accommodate audit-state changing between count and event queries. Hash-based indexing is supported to simplify locating a specific event, even in the presence of other events.

5.4.8.1. Audit history size

When querying current history size, the query may be submitted without payload, or must include a single 4-byte raw integer, with 0 as a value.

In both cases, element count is returned as a single 4-byte raw integer. Note that there is an upper bound on the returned element count, since the active history is stored in a circular buffer of fixed size.

5.4.8.2. Audit history entry

When requesting a specific audit event from history, the host must specify a particular entry through relative index or absolute state(hash). Both cases must supply a payload.

When indexing entries based on their position, a single 4-byte raw integer must be supplied, which is used as a one-based index.

As detailed in (5.4), there is an upper limit on returned event bytecount (currently, 290).

5.5. Structures used by extensions

5.5.1. Multi-level security (MLS) memory-typing extensions

MLS-extended data is augmented with attributes describing its type (currently) and is signed by the original module producing it. These structures replace plain, unauthenticated data for use by MLS-aware keys.

```
mlsData ::= SEQUENCE {
    version         OCTET STRING,
    data            OCTET STRING,
    attributes      OCTET STRING,
    nonce           OCTET STRING,
    signIdentifier  OCTET STRING,
    signature       OCTET STRING
}
```

Version is encoded as a 32-bit raw integer. Currently, the only supported version is 1.

Attributes are encoded as described in (3.2.), containing at least XCP_BLOB_MLS as a Boolean and CKA_IBM_MLS_TYPE as a type-identifing integer (3.2.3.).

In the current configuration, the encoding always writes attribute fields with the last 32 bits containing the CKA_IBM_MLS_TYPE value. Parsers, when encountering v1 results, may rely on this shortcut, if they do not intend to parse the entire attribute field.

The nonce has an internally enforced minimum length (not exported as a constant). It is inserted to prevent generating an MLS signature on entirely user-controlled data (i.e., provide a signing oracle).

The signIdentifier field contains a truncated key identifier, a mnemonic of the signing key (such as (6.7.1.)) but not that of the signing algorithm. We expect the MLS-aware context to unambiguously identify its signing keys, and the field is only used to simplify later lookups. It is ignored by the backend when MLS data is read back, since then the expected MLS-signing key should already be available.

It signIdentifier is inserted by the signing backend as a 32-bit truncated value, derivation specific to the signing key type. For the currently used (H)MAC keys, see (6.7.) for derivation algorithm.

The signature field contains a signature calculated across all preceding bytes, between the encapsulating SEQUENCE and tag+length fields of the signature OCTET STRING, inclusive.

5.5.2. TRNG entropy-pool log access

History of the TRNG entropy pool, if supported by the backend, is accessed as a file, and its slices are identified as file-part queries are (6.17). There is a single entropy pool, and it MUST be designated as file ID 0 [access to the entropy pool is through a file-like interface, but currently there is only one accessible pool].

The entropy pool is internally stored as a circular buffer; size query (6.17.1) returns the currently available number of bytes, but no other information. [In other words, test applications are expected to deal with rollover.]

To advance the TRNG pool explicitly, see the XCP_DEV_DRAIN_ENTROPY SYS-TEST service.

5.6. Host target and module management

Targets and target groups can be registered and deregistered with functions operating on the the XCP_Module data structure. Registering a target or target group will generate a target token.

All data types described in this chapter are specific to the reference implementation of the host library and not part of the wire format itself.

Other host libraries may use this description as a reference for their own target and module management.

5.6.1 Target token data type

Host libraries can employ the data type target_t to select which target should be addressed in a host library function call. The base type of target_t is implementation specific, however we use an integer base type to allow bit mask operations.

Domain specific flags are set on target tokens. The flags can be set and masked with normal bitwise operations after the target token was associated with a target (with m_add_module).
All domain specific flags are for diagnostic/development purposes only and only usable with m_DigestSingle:

XCP_TGTFL_WCAP	Capture a wire request in the output buffer (digest buffer) without sending it to a module.
XCP_TGTFL_WCAP_SQ	Size query: Return size of a request in the output buffer length field (dlen).
XCP_TGTFL_NO_LOCK	Ignore sequential locking upon functional request.
XCP_TGTFL_API_CHKD	Target's API has been checked.

Target tokens should be initialized to XCP_TGT_INIT before setting them.

5.6.2 The XCP_Module structure

The XCP_Module structure is used to maintain the module configuration of the systems. A target is represented in the structure through a combination of the module number and a domain in the domainmask.

5.6.2.1 Structure version

The version of the structure controls which features and fields of the structure are available. Version values lower than XCP_MOD_VERSION are allowed as long as the value is higher than zero, but functionality might be degraded. Version values higher than XCP_MOD_VERSION are not accepted.

The actual version value may be queried at runtime (see m_add_module)

5.6.2.2 Module flags

Flags control which fields are interpreted and how the host library interprets the XCP_Module structure. Some flags may enable fields that are only meaningful on some platforms. The host library accepts the following flags:

XCP_MFL_SOCKET	The backend is socket-attached.
XCP_MFL_MODULE	The backend is identified as a a number in an array-of-modules.
XCP_MFL_MHANDLE	The'mhandle' field is used to connect to a backend.
XCP_MFL_PERF	Performance statistics are collected for a module in the perf field.
XCP_MFL_VIRTUAL	The module is flagged as a virtual module, which is used to create a target group.
XCP_MFL_STRICT	Fail on correctable errors.
XCP_MFL_PROBE	Probe if target is reachable before registering it.
XCP_MFL_ALW_TGT_ADD	Allows to use a target in any functional and administrative call without registration.

The flags are further described in chapter 5.6.3

5.6.2.3 Domains and domainmask

The domains field indicates the maximum domain number that can be used on the system. It can be set to any value lower or equal the system default. If the domains field is set to zero the system default value is used. If no system default can be found the EP11 default value is used:

XCP_DOMAINS	256

The domainmask contains all domain indices that should or can be used (if target probing is enabled - see chapter 5.6.3.5) for this module. Domains can be set/unset in the domainmask with the SET_DOM() and CLR_DOM macro. The DOM_IS_SET() macro can be used to check if a specific domain is set in the domainmask.

The domainmask field itself is an array 32 Bytes in size. One bit in the domainmask represents one domain from MSB to LSB. So domain zero is bit eight (left-most bit) of the first byte (value 0x80). Domain 255 is the right-most bit of Byte 31 (value 0x01) in the domainmask field.

The domainmask field is updated by the host library in the registration and deregistration process to represent the actual state of the registered domains.

5.6.2.4 The module number

This field is platform dependent and enabled with the XCP_MFL_MODULE flag. The module_nr field is used for the module number which identifies a module in a array-of-modules configuration. See chapter 5.6.3.3 for more information.

5.6.2.5 The socket field

This field is platform dependent and enabled with the XCP_MFL_SOCKET flag. The socket field contains information needed for an HSM reachable through a socket connection. This field is development only.

5.6.2.6 The module handle

This field is platform dependent and enabled with the XCP_MFL_MHANDLE flag.
The mhandle field can be associated with a handle that points to
target information which is needed to route a request to a module.
The field is needed for platforms where the target management is configured
outside of the host library and the host library is only used to create
requests and parse responses. The field is restricted to development.

5.6.2.7 The perf fields

The perf field is enabled with the XCP_MLF_PERF flag. For every domain one byte
is reserved, which contains the performance value of the last request.
See chapter 8.2.1. for more information about these performance values.

5.6.2.8 API ordinal

The API ordinal of the module as specified by the api field of the XCP_Module
structure.
This field can either be zero or a positive integer.
By default the following checks will be performed before the first functional
request is sent to a module.
If api equals zero, an API query (section 1.1.2) is sent to the targeted module
to determine the value of api.
For target groups, the largest common API will be used to communicate with all
the modules in the group.
If api is a positive integer, no additional requests are sent to the module.
The value of api, in either case, is stored internally and used for
communicating with the respective module.
With a non-zero value, a particular API can be requested to be used for the
interaction with the module.

If XCP_MFL_PROBE is enabled for a module, then the request described above
will be sent as a result to a call to m_add_module (sec. 5.6.3.5).
For a single targeted module, m_add_module will throw an error if the requested
API ordinal is larger than that of the module or if the probe fails.
For a target group, the particular module will not be added to the group if the
probing fails or if the API of the module is incompatible with the requested
api.
In contrast to individual targets, if also XCP_MFL_STRICT is set, an error will
be thrown and no modules will be added to the target group.

If the API ordinal is set by the user and XCP_MFL_PROBE is not enabled, no
additional checks will be performed.
In case the API ordinal was determined automatically upon module addition,
the API ordinal is written back into the module structure, only if XCP_MFL_PROBE
was active during m_add_module call.

If the XCP_Module structure does not support the api field, an API ordinal of 2
is assumed. For empty target groups with an XCP_Module structure that contain
the api field, the most recent supported API ordinal is assumed.

See also section 1.1.1 for more details on the API ordinal.

5.6.3. Flag dependent host library behavior

The module flags control how the host library should interpret the
XCP_Module structure. Depending on the specified flags some fields may be usable
or not.

5.6.3.1 Fail on correctable errors

Correctable errors are treated as failures with XCP_MFL_STRICT. Chapter 5.6.4 lists the implications of enabling this flag. The flag does only affect registration and deregistration.

5.6.3.2 Virtual modules

Virtual modules are created by setting the XCP_MFL_VIRTUAL flag, which is platform independent. Targets registered for a virtual module create a new target group or are added to an existing target group. Without the flag only single targets can be registered. A virtual module can have as many targets as the module on this platform does allow (limit is XCP_DOMAINS). A virtual module with zero targets addresses all targets available for the system.

5.6.3.3 Backend types

The host library can handle different backend types.
Which type of backend a XCP_Module structure is supposed to describe is handled through flags:

XCP_MFL_MODULE	A value of the module_nr field identifies the backend in an array-of-modules.
XCP_MFL_SOCKET	The backend is socket-attached and uses the socket field.
XCP_MFL_MHANDLE	A handle is associated with the mhandle field containing target information transparent to the host library.

The fields enabled with these flags are described in more details in chapter 5.6.2. Refer to chapter 5.6.4.1 for more information on how the platform dependent behavior of these flags is implemented.

Host libraries need to be built for one of the known backend types. Please note that all backend types with the exception of the XCP_MFL_MODULE type are restricted to development only.

5.6.3.4 On the fly target adding

If XCP_MFL_ALW_TGT_ADD is active for a module, targets for these modules do not need to be registered with m_add_module, but can be registered implicitly by simply using target tokens in normal m_* calls. This is by default disabled and needs to be enabled for a module using the XCP_MFL_ALW_TGT_ADD flag.

5.6.3.5 Probing targets

If XCP_MFL_PROBE is enabled in an XCP_Module structure an API query (see chapter 1.1.2) is sent to every target that should be registered. Only targets that can be queried are registered, others are ignored. Please note that an unreachable target is a fatal error if XCP_MFL_STRICT is on.
Please see chapter 5.6.4.2 for more information.
If the api variable of the XCP_Module structure (sec. 5.6.2.8) is set to zero the determined API ordinal of the module is written back to the structure.

5.6.3.6 Performance measurements

If XCP_MFL_PERF is active for a module the performance category of an incoming response is extracted and saved to the perf field. See chapter 5.6.2.7. for more information.

5.6.4 Correctable and uncorrectable errors

When registering or deregistering a module the host library can detect errors in the module data structure and in most cases correct them. If XCP_MFL_STRICT is ON all correctable errors are turned into a return code instead.

The following error situations are detected and corrected when the flag is not ON.

5.6.4.1 Backend handling errors

Backend flags that are not supported by a host library are ignored as long as the minimum number of flags for the given platform are supplied. For example the IBM Z platform needs the XCP_MFL_MODULE flag. Any other backend flag is ignored. If the strict flag is ON backend flags not supported by the host library are treated as an error.

A use case is to provide information about different platforms in one module structures. All flags and fields that do not conform to the current platform are ignored by the host library. So it is possible to implement a single program without relying on platform dependent code.

5.6.4.2 Probing unreachable targets

If probing a target fails the error can be recovered if another domain is usable in the domainmask. This only works for virtual modules and allows to create a target group without knowing which targets are actualy available on the system. This is not possible when in strict mode.

5.6.4.3 Domain index correction

If the domains field is larger than the maximum value obtained from the OS the domains field is limited to the maximum value from the OS. Changing the domains field might result in invalidating targets registered already, which is why this is not possible in strict mode.
Domains not honoring the domains field are removed from the domainmask. If no domains are left an error is reported. In strict mode not honoring the domains field is an error.

5.6.5 Host specific return codes

Host libraries implement host specific return codes for management functions that do not relate to any PKCS#11 functionality. See the EP11 header file for a complete list of supported return codes. General return code names start with XCP_* and return code names related to module management have the prefix XCP_MOD_*.

6. Primitive structures

6.1. Source partition/VM

Originating sources, partitions/VMs are indicated in a dedicated field. We use 32-bit fixed-size fields, both in CPRBs (section 1.3.1) and within OCTET STRINGs. The field within OCTET STRINGs is a fixed-size raw integer, i.e., without an ASN.1 INTEGER tag.

Note that we do not realistically expect to encounter 2^32 sources, but reserve such a size to be able to accommodate a growing number of sources without format changes.

Backends ignore the source identifier and return it without modification. It is present within CPRBs to facilitate convenient access-control filtering on the host, without parsing the command or payload.

6.2. Target domain

Domains are encoded as fixed-size, 32-bit, big-endian integers. Note that as with targets, we do not expect to use 2^32 domains in the foreseeable future, but reserve such a size to be able to accommodate a growing number of domains without format changes.

Domain identifiers within ASN.1/BER structures reside within OCTET STRINGs of fixed size (4 bytes). CPRBs and other transport structures contain the same domain value without any encapsulation.

Card-level requests (all are administrative) must use 0 as a domain number. All requests are unambiguous, and one can always distinguish card-level requests from domain ones--therefore, whether the entire card or domain 0 is targeted is always clear.

6.2.1. Administrative domain

When targeting domains with administrative traffic, or responding with a domain number, we use a domain index (6.2) concatenated with an _instance identifier_ to form a unique domain number, and refer to this construct as "administrative domain". The domain instance identifier does not persist across zeroization, preventing replay of administrative traffic, even if the domain is reloaded with the same administrators after zeroization.

The domain instance identifier is a uniformly randomly generated 32-bit integer. We neglect the chance of collisions, since zeroization is an administrator-controlled activity, not assumed to occur sufficiently many times to become a problem.

Instance identifiers are generated upon initialization, for each domain--even if they are still not populated. They are replaced with newly generated random values during zeroization.

Administrative queries--if the domain field is present--and commands without signatures must use 0 as instance identifier. The correct value is always returned in a response. Card-level commands return 0 for both originating domain and instance identifier.

6.2.2. Session identifier

Session identifiers are used in blob-specific key derivation (in a host-opaque way, which we document elsewhere but not detail here), if the host library supports sessions. The identifier is a fixed-size binary value (32 bytes, XCP_WK_BYTES).

For objects not bound to sessions, the session identifier is all zeroes. The identifier-derivation process guarantees that nonzero identifiers are returned for all sessions.

Session identifiers are guaranteed not to have 0x30 as their first byte. This allows a single-byte check to differentiate between blobs starting with session identifiers, and MACed SPKIs, which may be used as blobs under other conditions.

Session identifiers are also included in PIN blobs (3.4).

6.3. Module identifier

The module identifier restricts administrative traffic to particular (instances of) particular modules. In requests, it must be empty for queries. It must be present for commands, including those without signatures; the host-provided counter is ignored for the latter.

Module identifiers are encoded as fixed-size, 16-byte values. The field concatenates a module-specific serial number (8 byte, XCP_SERIALNR_CHARS) with an instance-specific value (same size).

We fill the module-specific serial number based on card properties, as reported by card hardware. This value persists when firmware is reloaded.

The instance-specific value is randomly generated when firmware is loaded to the module, or when the card is zeroized.

We require a reasonable RNG generating this value---inside the backend---and do not consider collisions a potential problem.

The module identifier is always populated in responses.

6.4. Administrative transaction counter

The transaction counter matches the size of the internally maintained transaction state, as a fixed-size counter (16, XCP_ADMCTR_BYTES).

There is a counter for card-level commands, and a separate one for each domain. Counters are reset to zero upon zeroization. Wraparound is not supported: once transaction counters reach their possible maximum (ff...ff), the card may no longer be actively managed. (The size of the counter makes this infeasible under normal operations.)

Administrative queries may provide counters, or leave the field empty. Commands without signatures must provide transaction counter fields---but the provided counter is ignored. The current value---after the update, if applicable---is always returned in a response.

6.4.1. Transaction counter queries

We do not provide standalone queries for transaction counters, as the counter is included in all responses. In order to fetch it, one could issue a card/domain administrator list query--or some other administrative query which is always available--and retrieve the counter from the response.

6.5. Imported keyparts

During WK import, one or more keyparts are enveloped within the payload of a single xcpAdminReq structure (4.1). Each keypart is a standalone xcpAdminReq, containing a RecipientInfo as ‘‘payload’’ of its command, targeting the current importer. Administrator signature(s) must authenticate each the command block.

The enveloping xcpAdminReq is unsigned, therefore its signature field remains empty. Its targeting information must match that of the enveloped .

Each RecipientInfo encrypts one raw keypart (6.5.1) for the current importer.

Command blocks in each embedded xcpAdminReq must contain the same recipient identification as well as auxiliary information (admFunctionId of ‘‘import WK’’ (XCP_ADM_IMPORT_WK), domain, moduleIdentifier, transactionCtr). Auxiliary fields with each keypart must match that of the enveloping final xcpAdminReq. Note that the external xcpAdminReq itself lacks signatures, but its auxiliary fields must still be populated, and match that of the constituent keyparts.

6.5.1. Imported raw keyparts, without reassembly threshold

If keyparts are all combined through XOR, each encrypted keypart concatenates the following fields as plaintext without further formatting or padding:

1.	raw key(part) bytes	(32 bytes, XCP_WK_BYTES)
2.	VP of reassembled key, optional	(32 bytes)

The backend may infer the choice purely based on plaintext size, which is unambiguous.

The VP of the reassembled key is missing if keyparts were created independently. It must be supplied if the parts were created by a keypart export. All parts in a single Import WK command must possess or lack key VP; this policy must be enforced by host tooling collecting keyparts (note: it may not be directly observed by administrators, must be integrated into keypart-encryption procedures).

6.5.2. Import raw keyparts, N-of-M reassembly

If keyparts are combined through a threshold scheme, the plaintext within each encrypted keypart contains an additional field, the index of the keypart:

1.	index of keypart	(4-byte raw integer)
1.	raw key(part) bytes	(32 bytes, XCP_WK_BYTES)
3.	VP of reassembled key, optional	(32 bytes, XCP_CERTHASH_BYTES)

Due to the fixed sizes used, the presence or absence of reassembled VP may be unambiguously determined. Threshold-based keyparts may also be unambiguously separated from XOR-based ones.

Since a threshold scheme can always supply the VP of the reassembled key, it is strongly suggested that keyparts combined through a threshold always include the full VP. As with keyparts reassembled without use of thresholds (6.5.1), all or none of the individual keyparts must include the VP of the reassembled key.

6.6. Exported keyparts

```
exKeyParts := SET OF exKeyPart

exKeyPart := RecipientInfo
```

Exported keyparts are concatenated and signed as a compound response in the payload (xcpAdminRspBlk), and not individually signed. Each RecipientInfo contains "exported raw keyparts" (6.6.1) as its encrypted content, with one targeting each receiving KPH.

The RecipientInfos of exported keyparts are in arbitrary order, unrelated to the order of certificates in the originating "export WK" command.

6.6.1. Exported raw keyparts

Raw keyparts concatenate the following fields without further formatting:

```
1.  index of keypart, optional        (4-byte raw integer)
                                      (present only if exported through
                                       threshold scheme)
2.  raw key(part) bytes               (32 bytes, XCP_WK_BYTES)
3.  VP of the keypart                 (32 bytes)
4.  VP of the originating key         (32 bytes)
```

VP of the originating key and the keypart are both present, and are identical, when the key is transported as a single entity.

RSA-encrypted export contains the entire payload encrypted by RSA: the shortest supported modulus is sufficiently large to contain the entire payload.

EC-encrypted payloads are encrypted by a session-unique symmetric key, which itself is transported through regular one-pass ECDH.

6.7. Key verification patterns (VPs)

To calculate the verification pattern of a symmetric key, calculate the SHA-256 hash of its concatenated, single-byte type and the raw key bytes. The resulting 256-bit hash identifies the symmetric key (see XCP_KEYCSUM_BYTES, 32, as the constant for size), with the trailing (least significant) 32 bits set to all-zeroes.

For AES-256 keys, we use 0x01 as type, therefore the verification pattern is SHA_256(01 || <raw_key>), with the last 4 bytes set to zero.

VPs are intentionally made ambiguous by forcing some of their bits to a fixed value. This change inhibits fully deterministic dictionary-building to recover WKs, while not inhibiting regular use (at least, with negligible probability). Since WK mismatches are resolved unambiguously through exact matching, discarding bits of even full WK VPs has no security implications.

We currently only use AES-256 keys, and do not define other key types.

Key parts inherit the type of the key they have been separated from. As an intentional side effect, the VP of a single-part key and its single "keypart" are identical.

6.7.1. Wrapping key identifier (WKID)

A shorter version of VPs, WKIDs identify WKs through their truncated hashes. We use a fixed-size value (16, XCP_WKID_BYTES), the most significant bytes of the verification pattern.

6.8. PKCS#11 structures

For the list of PKCS#11 constants, see pkcs11.h (specifically, pkcs11t.h).

6.8.1. Mechanisms

Mechanisms are serialized as at least 4-byte buffers, with the leading 4 bytes containing the 32-bit mechanism value (CKM_...) as a fixed-size raw integer. Mechanisms without parameters must contain only these bytes. Parameters follow the mechanism bytes without any intermediate padding.

6.8.1.1. Mechanisms with IV

Mechanisms with IVs follow the mechanism with the IV bytes, of fixed size, as specified by the cipher.

The current list of IV mechanisms and their IV bytecount is:

```
  - CKM_AES_CBC              [16]
  - CKM_AES_CBC_PAD          [16]
  - CKM_DES3_CBC             [8 ]
  - CKM_DES3_CBC_PAD         [8 ]
  - CKM_DES_CBC              [8 ]
  - CKM_DES_CBC_PAD          [8 ]
  - CKM_IBM_CPACF_WRAP       [16]  -- vendor extension
```

6.8.1.2. RSA-PSS and OAEP

RSA-PSS mechanism parameters follow the mechanism with three concatenated big-endian, 32-bit raw integers, without further formatting or padding. Parameters are in the following order:

1. hash mechanism
2. mask generation function (mgf) (CK_RSA_PKCS_MGF_TYPE)
3. salt bytecount

Fields correspond to the PKCS#11 CK_RSA_PKCS_PSS_PARAMS structure. Hash algorithms are currently limited to SHA variants (CKM_SHA_1, CKM_SHA224, CKM_SHA256, CKM_SHA384, or CKM_SHA512). For unhashed PSS variants, a valid mechanism from the list must be present; hashed PSS mechanisms may supply the special-purpose ‘‘PSS default’’ value (0xffffffff).

In our implementation, the MGF hash function used must match the data-digesting function (CKG_MGF1_SHA1, CKG_MGF1_SHA224, CKG_MGF1_SHA256, CKG_MGF1_SHA384, or CKG_MGF1_SHA512), or contain the ‘‘XCP_PSS_DEFAULT_VALUE’’ 32-bit value 0xffffffff.
The default constant has been selected not to be possible under normal use; it derives the mask-generation function from the mechanism (or the hash function---first parameter---if the mechanism is an unhashed one).

Salt bytecount is currently limited to digest bytecount, or 0 if no salt is to be included (see notes under section 9.1., EMSA-PSS, in RFC 3447, for the restricted set of possible sizes). As a special case, the 32-bit value XCP_PSS_DEFAULT_VALUE (0xffffffff)---out of range of all reasonable hash functions---implies using a salt size identical to that of the hash function.

Salt bytecount is currently ignored during signature verification:

```
bytecount is derived from the signature itself.

Since hashed PSS mechanisms already prescribe a hash function,
a wire structure containing three copies of  0xffffffff forces
standard-recommended defaults for PKCS1 v2.1 PSS signatures for
that specific hash function.

Note that keeping MGF and data-digest function in sync is prudent,
prohibiting attacks on PSS by mismatched hash functions, (cf. Section 8,
Security considerations, in RFC 4055, and Section 8.1 in RFC 3447).

The wire encoding of PSS structs is transparently managed by our host
library, which accept regular CK_RSA_PKCS_PSS_PARAMS of PSS-aware
mechanisms at their .h interface, and serialize them to our wire
format. The only nonstandard extension is the default value, marked as
vendor-extended, which has no standard equivalent.

RSA-OAEP mechanism parameters follow the mechanism and consist of three
concatenated big-endian, 32-bit raw integers followed by an ASN.1 octet string
of specified length without further enveloping, formatting or padding.
Parameters are in the following order:

1.  hash mechanism
2.  mask generation function (mgf)  (CK_RSA_PKCS_MGF_TYPE)
3.  source (encoding parameter type, zero if no encoding parameter)
4.  encoding parameter (octet string)

The hash mechanism, mask generation function and source fields correspond to
the PKCS#11 CK_RSA_PKCS_OAEP_PARAMS structure. Both hash algorithm and mask
generation function must employ the same hash function (CKG_MGF1_SHA1,
CKG_MGF1_SHA224, CKG_MGF1_SHA256, CKG_MGF1_SHA384, or CKG_MGF1_SHA512).
In addition to the PKCS#11 standard, we define a set of equivalent mask
generation functions that employ SHA-3 algorithms (CKG_IBM_MGF1_SHA3_224,
CKG_IBM_MGF1_SHA3_256, CKG_IBM_MGF1_SHA3_384, CKG_IBM_MGF1_SHA3_512).
For the corresponding SHA-3 definitions see section 8.7.3.
The source field specifies the type of the encoding parameter.
Currently only no encoding parameter (source == 0) or octet strings are
supported (source == CKZ_DATA_SPECIFIED).
The same encoding parameter has to be passed to the decrypt operation that has
been used for the encrypt operation or it will fail otherwise.

6.8.1.3.  Custom ID-card (EAC) related functions

We define a single custom mechanism to support Extended Access Control
algorithms (CKM_IBM_EAC), and parameterize the mechanism to provide a
single entry point for EAC-related operations. Currently supported EAC
operations are selected by a single integer, which is the mandatory only
parameter.  The mechanism takes a single CK_ULONG parameter
in the reference C implementation.

In wire form, the EAC mechanism must be followed by a 32-bit raw
integer, its parameter. Supported values are enumerated under EAC_Var_t
definitions. As with other enumerated constants, the list may grow in
the future, without changing the meaning of previously defined values.

6.8.1.3.1.  EAC sub-variants

The current list of supported EAC constants is:

EACV_IBM_KEK_V101        1   -- secret -> secure messaging KEK (EAC v1.01)
```

```
EACV_IBM_MACK_V101       2   -- secret -> secure messaging MACK (EAC v1.01)
EACV_IBM_PWD_V200        3   -- passphrase -> secure messaging KEK (EAC v1.01)
EACV_IBM_HKDF            4   -- HKDF( key, salt ) -> PRF stream [RFC 5869]
                             -- salt is supplied with mechanism parameter
                             -- output bytecount specified as attribute
                             -- of derived key
EACV_IBM_BCHAIN_TCERT0   5   -- blockchain: derive tcert from base EC key
                             -- and additive cleartext [potentially insecure]
                             -- additive term must be supplied as aux.data
                             -- restrictions: base EC key must use 256+ bit
                             -- prime. Curve/type restrictions are inherited
                             -- from control points, not separately by mech.
```

If the EAC key derivation function is called with a nonce or salt,
the nonce/salt must be passed as the parameter to DeriveKey.
Currently, only the HKDF derivation option supports salting.

6.8.1.3.1. HKDF parameters

The HKDF key-derivation function MUST include a hash-function
identifier, and two optional parameters [‘‘info’’ and a salt], which are
a property of the HKDF process itself, not that of the base key. [See
RFC 5869, sections 3.1 and 3.2]

Parameters MUST be serialized as the following BER structure:

```
SEQUENCE {
    OCTET STRING hashfunction,
    OCTET STRING salt,
    OCTET STRING info
}
```

The hash function identifier MUST specify a PKCS11 hash function
constant or a non-GENERIC HMAC one.

6.8.1.4. ECDH1_DERIVE

CK_ECDH1_DERIVE_PARAMS structures are serialized in a BER structure,
containing the following fields:

```
SEQUENCE {
    OCTET STRING kdf,
    OCTET STRING sharedData,         -- possibly empty
    OCTET STRING publicKey
}
```

The key derivation function ‘‘kdf’’ is stored as a 4-byte raw integer.
Currently supported values are hash functions as given in the PKCS11 (v2.40)
standard. Shared data may be empty. CKD_NULL does not use shared data.

This structure is supported starting from API ordinal 3.
Modules using an older version, can still derive keys by using the raw key
material instead of the above sequence.
The resulting key will be the raw secret from the ECDH key derive without any
KDF applied, equivalent to using CKD_NULL.

6.8.1.5. CKM_IBM_CMAC

This vendor mechanism is used for CMAC signing and verification. It may
be used by sign/verify-capable AES and DES symmetric keys, and

does not take parameters.

6.8.1.6. DSA parameters

When generating DSA keys or parameters, an intermediate structure stores parameters (P,Q,G). This structure is returned when DSA parameters are generated through GenerateKey (using CKM_DSA_PARAMETER_GEN as mechanism).

When calling GenerateKeyPair to create a DSA keypair for specific parameters, the vendor attribute CKA_IBM_STRUCT_PARAMS must contain the full P,Q,G structure. The content must be the full ASN.1/BER SEQUENCE enumerating P,Q,G parameters (7.8), or a raw packed alternative storing raw P,Q,G values (7.8.1).

Note that the vendor attribute is shared with DH key generation, decoding unambiguously from the mechanism.

6.8.1.7. PBE parameters (CKM_PBE_... mechanisms)

Password-based key derivation turns a host-contributed password and salt into a key and an initialization vector (IV).

	offset	field	bytes	notes
		fixed header		
1	0	PBE type	4	0: clear password 1: encrypted password
2	4	password bytecount	4	verbatim if clear secret-key blob if encrypted
3	8	salt bytecount	4	up to 256 bytes
4	12	iteration count	4	up to 65536
		variable size fields		
5	16	password	(dynamic)	see bytecount above
6	...	salt	(dynamic)	see bytecount above

Fields are big-endian.

A PBE type of 1 is reserved for passwords within secret-key blobs. The password field must then contain a secret-key blob, possibly a generic secret key, with CKF_GENERATE enabled.

Password and salt are concatenated without further formatting or padding. Bytecounts are limited to fixed values (see XCP_PBE_PWD_MAX_BYTES and XCP_PBE_SALT_MAX_BYTES). Passwords are accepted in raw form, the caller is responsible for BMPstring expansion (if strict compatibility with PKCS12 is required).

The backend may limit iteration counts, returning CKR_MECHANISM_PARAM_INVALID if the requested value is over the limit of 65536 (XCP_PBE_ITER_MAX).

PBE mechanism-parameter structures are by construction unambiguously distinguishable from ASN.1/BER structures. We may therefore add BER representations in the future without impacting existing applications.

The IV derived from a PBE structure is returned concatenated with the PKCS11 checksum of the derived key (7.6.1).

6.8.1.8. SPKI MAC conversion (CKM_IBM_TRANSPORTKEY)

```
This mechanism is used to turn a raw SPKI into a MACed one (7.4.1).
It uses no parameter, obtaining all necessary information from the SPKI
itself, and attributes provided with the call to UnwrapKey().

The UnwrapKey() output of this mechanism is the MACed SPKI, and the
truncated SKI of the key as a checksum (7.6.3), extended by the
size-reporting tail used by UnwrapKey() (7.6.1)

6.8.1.9.  DH parameters

When generating DH parameters, an intermediate structure stores prime
and generator (P,G). This structure is returned when DH parameters
are generated through GenerateKey (using a DH parameter-generating
mechanism).

When calling GenerateKeyPair to create a DH keypair for specific
parameters, the vendor attribute CKA_IBM_STRUCT_PARAMS must contain the
structure describing P and G. The content must be the full ASN.1/BER
SEQUENCE enumerating P,G parameters (7.8), or a raw packed alternative
storing raw P,G values (7.8.2).

Note that the vendor attribute is shared with DSA key generation,
decoding usage unambiguously from the mechanism.

6.8.1.10.  DH key derivation

The PKCS3 DH key derivation mechanism, CKM_DH_PKCS_DERIVE, uses a single
parameter, the DER-encoded public key of the other party.  The only
bytes appended to the mechanism must be that of the public key ASN.1
SEQUENCE, without further formatting or padding.

Parameters used by CKM_X9_42_DH_DERIVE use a packed structure identical
to that used by CK_ECDH1_DERIVE_PARAMS (6.8.1.4).

6.8.1.11.  Mechanisms with variable-sized results (..._GENERAL)

Mechanisms which may select output size, when requesting output, must
append the requested number of bytes as 32-bit raw integer as the single
mechanism parameter. This value corresponds to the CK_MAC_GENERAL_PARAMS
PKCS11 parameter.

When supplying results to such a mechanism, the same value needs to be
provided, even if redundant, as we may observe the result size. (We may
relax this requirement in the future.)

Mechanisms following this convention are the following:
    standard mechanisms:
        CKM_SHA224_HMAC_GENERAL       (0x00000257)
        CKM_SHA256_HMAC_GENERAL       (0x00000252)
        CKM_SHA384_HMAC_GENERAL       (0x00000262)
        CKM_SHA512_HMAC_GENERAL       (0x00000272)


SECURITY NOTICE: we consider allowing truncated signature GENERATION but
not VERIFICATION: allowing verification of truncated signatures with
user-specified lengths allow users to reconstruct longer signatures, if
they have access to a shorter, valid signature (see security rationale).
Note the corresponding control point, which allows verification of
truncated signatures.
```

6.8.1.12 Protected key import

The CKM_IBM_CPACF_WRAP WrapKey mechanism is used to import a key (blob) flagged as XCP_IBM_PROTKEY_EXTRACTABLE as an protected key. The key encryption key is provided by the system firmware. System firmware can restrict this request, to only send it in a privileged environment. Currently AB keys are not supported with this mechanism.

EP11 returns the wrapped key as an octet string which contains a special wrapped key format:

```
                               bytecount      note
    -----  MACed:  -----------------------------------------------------
1.  structure version          2              0x0001
2.  KEK identifier             16             XCP_WKID_BYTES, see (6.7.1)
3.  key type                   4              wrapped key type
4.  Bit length                 4              length of the key in bits
5.  IV                         16             random IV
    -----  encrypted:  -------------------------------------------------
6.  wrapped key                96
    -----  end of encrypted region  -------------------------------------
7   NULL padding               54             padding of null bytes
    -----  end of MACed region  -----------------------------------------
8.  MAC                        32             covers all preceding bytes
                                              (incl. clear header)
```

System firmware may transform this to the following format:

```
                               bytecount      note
                               ---------      ----
1.  structure version          2              0x0001
2.  reserved                   16             reserved 0
3.  key type                   4              wrapped key type
4.  Bit length                 4              length of the key in bits
5a. Token size                 8              size of the token + the
                                              verification pattern
5b. reserved                   8              reserved 0
    -----  pro begin:  -------------------------------------------------
6.  wrapped key                var
7.  verification pattern       32
    -----  Token end:   -------------------------------------------------

7   NULL padding               var            padding of null bytes to
                                              224 bytes. Making the structure
                                              the same size as sent by EP11.
```

The firmware generates a token which acts as the protected key.
EP11 supports the following key types:

```
AES with bit length 128, 192 and 256            = 0x1
2DES and 3DES                                   = 0x2
EC-P with bit length 192, 224, 256, 384 and 521 = 0x3
ED25519 and ED448                               = 0x4
```

Supported key types and bit lengths vary between different systems and are also restricted by the EP11 control point set up. Consult your architecture for key types and bit lengths which can be used on your system.

EP11 flags domains which can be used for protected key import (wrapping key is

set and domain has the correct policy set up) with CK_IBM_DOM_PROTKEY_ALLOW.
See the CK_IBM_XCPQ_DOMAIN query description for more information.

The MAC key is derived from the key encryption key and the MAC mechanism is HMAC based on SHA-256. The derivation mechanism uses the TK concatenated with information that identifies the MAC key, prefixed with the number one and hashed with sha256: sha256(be32(1) || <key> || <kdf_info>)}.

The key information is described in a ASN.1 sequence defining the used algorithms and i390CC and EP11 as the owner of the key material:

```
kdf_info ::= SEQUENCE {
 PartyUInfo (BIT STRING, "I390CC")
 PartyVInfo (BIT STRING, "EP11")
 AlgorithmID (OID, hmacWithSHA256)
```

[section excluded from published version]

6.8.1.19. Post-quantum algorithms: CKM_IBM_DILITHIUM variants

As an experimental feature, the module is capable of supporting the round 2 version of the ‘‘Dilithium’’ digital-signature algorithm in a restricted setting; support is indicated by the presence of CKM_IBM_DILITHIUM (0x80010023) in the mechanism list.
This mechanism is used for keypair generation and for sign/verify operations.
Incremental sign/verify operations with SignUpdate/VerifyUpdate are not supported.

While Dilithium itself is a scalable algorithm, for application/compatibility reasons, we only support the ‘‘high-security’’ version of round 2 Dilithium, classified as Category IV in the original NIST PQ RFP [4.A.5. Security Strength Categories]. This version is called ‘‘Dilithium-1536x1280’’ or ‘‘Dilithium-6-5’’ in the specification, and this version remains the default for future releases.

When generating new Dilithium keys, the wire-encoded mechanism without additional parameters implies use of the default version. Future versions, when they add support for other key sizes, will optionally allow designating an object identifier (OID) with CKA_IBM_PQC_PARAMS (0x8001000e) which selects a size other than the default one.

Dilithium-related calls other than key generation, including UnwrapKey(), infer Dilithium sub-variant size from the key object, and require no mechanism parameters. (The only other import-like function, DeriveKey(), can not create new Dilithium keys.) Only the key type needs to be specified as CKK_IBM_PQC_DILITHIUM (0x80010023) with the attribute template for UnwrapKey

See pq-crystals.org for algorithm specifications.

6.8.1.20. Montgomery and Edwards based elliptic curves

Contrary to other curves Montgomery and Edwards curves are restricted to specific mechanisms for sign/verify and derive operations and can therefore not be used with the generic ECDSA and ECDH mechanism. Every curve has separate mechanisms that match with the OID of an Edwards/Montgomery curve. See section 8.1.1.5.1. for the list of corresponding OIDs and mechanisms and see section 6.8.1.4. for the mechanism parameter of the ECDH mechanisms. The ECDH mechanisms return the derived key as an EP11 blob back to the user. An exception are the raw mechanisms (CKM_IBM_EC_X25519_RAW and CKM_IBM_EC_X448_RAW), which return the key encrypted under a KEK.

Certain attributes are rejected when specified in key generation templates as the associated operations are not supported/possible on these curves.

Prohibited for Edwards curves:

CKA_DERIVE

Prohibited for Montgomery curves:

```
CKA_SIGN -- CKA_VERIFY is silently ignored in public key templates, as we can
            not control public keys
```

We will revisit all mechanisms regarding these curves as they get standardized in the PKCS#11 standard.

6.8.2. Mechanism list

The mechanism list reported by GetMechanismList() is static, containing all mechanisms technically supported by the backend. The list may further be restricted by dynamic settings---such as control points, indirectly by compliance setup---but this is not reflected in the list.

6.8.3. Attributes

6.8.3.1. Compliance attributes

Standards' compliance, when encoded as an attribute for a key, is---currently---stored in a 32-bit raw integer storing a 32-bit bitfield, one of the vendor-specific attributes (CKA_IBM_STD_COMPLIANCE1).

Any number of bits may be supplied when the attribute is encoded for the wire. The backend removes any unsupported bits, therefore, host code should compare the generated object, and react to any unsupported bits (those which are not supported by the backend).

Blobs tied to certain mode(s) are rejected when they are sent to a domain with more permissive settings. Note that compliance mode is reflected in the clear blob header.

The list of currently supported compliance bits is under (8.1.1.3).

6.9. Raw integers

Integer-type values, such as indices or sizes, are sometimes written into OCTET STRINGs or similar structures. We refer to such values as 'raw integers', meaning a minimal-length big-endian encoding of the corresponding integer.

In some cases, raw integers are forced into fixed-size fields, or are padded to some minimum length. Since this is the exception, we mention it as a special case.

6.10. Empty fields

We allow fields to be completely missing, such as when some parameter is optional. These fields are encoded as zero-length, without a Value field, as valid--but somewhat unusual--structures. The rationale is to simplify optional fields: instead of designating fields optional, we allow them to remain empty, and keep the parser simple. Specifically, without optional fields, we'll always know how many OCTET STRINGS must be present in SEQUENCEs, and do not need CONTEXT-SPECIFIC tags to identify optional fields.

Note that empty fields differ from raw integers storing zero: the length of encoded zeroes is always positive, either 1 (minimal encoding) or larger (fixed or minimum-size fields).

6.11. Date/time fields

Date/time is passed as fixed-size UTC time, formatted as ASCII digits into a 16-byte packed form "YYYYMMDDhhmmss00" (4-digit year, one-based month, one-based day, hour, minutes, seconds, and two mandatory zeroes). No trailing zero or other padding is used.

Note that this UTC format is identical to the one present in the PKCS#11 CK_TOKEN_INFO structure.

6.11.1. Incremental date/time

For services which allow incremental time change, such as the incremental form of administrative set-time (XCP_ADM_SET_CLOCK), date/time fields MUST be prefixed by an ASCII '+' or '-' character (0x2b or 0x2d, respectively). Individual time units of the date/time field are then interpreted as relative shifts.

Note that an incremental shift without changing time is valid, therefore a field with leading +/- and all-(ASCII)0 date/time is possible. [These unusual forms are used, for example, when an unauthenticated time-shift command is sent to prevent subsequent unauthenticated updates until the requisite amount of time has passed.]

6.11.2. Audit record date/time

Audit records, optimized for small bytecount, store UTC date/time in a packed 48-bit field, a derivative of UNIX time_t, extended to millisecond precision, and safe from the Y2038 problem (i.e., overflow of 32-bit time_t in 2038).

The 48-bit field concatenates two values, the number of seconds as a 36-bit integer in its most significant bits (time_t base, i.e., number of seconds since 1970-01-01T00:00:00), and milliseconds as a 12-bit integer.

Simply shifting the 48-bit integer by 12 bits turns the field into a time_t second. This format has been selected to allow easier parsing directly from audit hexdumps, as opposed to a more efficient encoding which would require explicit decoding of the 48-bit field.

Note that 36-bit time_t seconds wrap (first) in Year 4147, which must be tolerated by applications, and will not be fixed in the backend. (Note that 4147 is beyond the planned retirement date of all current developers.)

6.11.3. Performance-test date/time

Performance test code, an optional feature, uses a derivative of audit-record timestamps (6.11.1), storing time with higher resoluton in a packed 64-bit field, concatenating the least significant 32-bits of time_t seconds with a 32-bit field storing nanoseconds. Since performance tests are not assumed to be impacted by rollover of time_t, we acknowledge the 32-bit time_t limitation (i.e., the Y2038 problem), and depend on users reconstructing time across rollovers.

As with audit records, ease of parsing justifies the minimal loss in accuracy, therefore the separate field for sub-seconds. While the sub-second field is capable of reporting events with nanosecond resolution, its resolution is limited to that of the backend.

6.12. Certificate index

Querying specific (OA) certificates requires an index. When present, we encode the value as a zero-based raw integer, counting backwards from the current (latest) certificate. The parent of the current certificate is encoded as 1, its parent as 2, etc.

6.13. Query structures

Functional queries are available through the get_xcp_info call. They are parametrized through a query type, defined as one of the CK_IBM_XCPQUERY_t constants (see 8.7.1).

Two input parameters must be supplied to the query, both as fixed-size 32-bit raw integers. The first parameter specifies query type, one of the types enumerated in CK_IBM_XCPQUERY_t. The second parameter may be used by the query, and its interpretation is query-dependent; currently, none of the queries uses it (we may add selective selftests as the first use). For query types which do not require the parameter, the value must be zero.

See (5.1) for a listing of compound structures returned by queries.

6.13.1. Extended flags

The following card-level extended flags are defined for capabilities beyond standard PKCS#11:

CKF_IBM_HW_EXTWNG	1	Module monitors removal from its slot. See also XCP_ADMM_EXTWNG under admin attributes. Partition-based instances do not set this bit.
CKF_IBM_HW_ATTEST	2	Module features hardware-assisted attestation, authenticated by an authority in secure hardware (such as outbound authentication on IBM HSMs).
CKF_IBM_HW_BATTERY	4	Module has a battery, and may raise warnings about low battery (8.1.1.2)
CKF_IBM_HW_SYSTEST	8	Module supports---insecure---test functions
CKF_IBM_HW_RETAIN	16	Module may retain hardware-internal keys
reserved	32	Reserved for future use
CKF_IBM_HW_AUDIT	64	Module supports audit-event logging
reserved	128	Reserved for future use
reserved	256	Reserved for future use
CKF_IBM_HW_ADM_DOMIMPORT	512	Module supports domain import See also related extended capability (CK_IBM_XCPXQ_DOMIMPORT_VER)
CKF_IBM_HW_PROTKEY_TOLERATION	1024	module tolerates blob attributes

```
                                     related to the protected-key capability
                                     see also CKA_IBM_PROTKEY_* description
```

These system-specific additions mainly mirror hardware features specific to IBM HSMs, and therefore needed to define capability bits outside PKCS#11 scope. Lack or presence of these extended features generally does not change PKCS#11-visible functionality, it interacts with infrastructure we add out of PKCS#11 scope.

6.13.2. Domain flags

The following bits are defined within domain flags:

```
CK_IBM_DOM_ADMIND           1  -- administrators present
CK_IBM_DOM_CURR_WK          2  -- current WK is present
CK_IBM_DOM_NEXT_WK          4  -- pending/next WK is present
CK_IBM_DOM_COMMITTED_NWK..  8..-- pending/next WK has been committed
CK_IBM_DOM_IMPRINTED       16  -- left imprint mode
CK_IBM_DOM_PROTKEY_ALLOW   32  -- policies allow protected key
```

6.13.3. Function Control vector (FCV)

When queried, only parts of the FCV structure are returned to the host.

A backend without a loaded FCV reports a default value, which has no functionality enabled. During development, we generally load a default which enables everything; production code manages FCVs properly.

See the non-wire portion of the structure document for FCV layout details. Inheriting the layout from another specification, we do not replicate those details here.

6.14. Firmware identifier (FWID)

During building, a hash is calculated over the XCP code and the backing CSP. The value is stored as a binary field of XCP_FWID_BYTES, and is constructed in an opaque manner (may be a truncated version of a longer hash).

We intentionally do not publish details of the hash calculation, and may change details without notification. The identifier will always be calculated by an accepted cryptographic hash function, but the exact input form may depend on build parameters (therefore the opaqueness). Input to the generated hash always includes the current state & setup of the XCP source, and a binary form of the underlying CSP, but in an order or form which we may change.

If a truncated version of the identifier is used, we take the most significant bits (i.e., leading bytes). As with PKCS#11 key checksums (7.6), we acknowledge--but do not try to detect--collisions. We only provide a weaker guarantee: firmware versions with differing truncated FWIDs are guaranteed to be different.

6.15. Domain mask

When identifying a number of domains simultaneously, such as when querying the aggregate compliance of a set of domains, the backend takes a domain mask as a parameter. The same domain-mask format is used when the host requests a specific set of domains, such as when exporting state. We support a simple zero-based enumeration of domains, or an alternate form selecting a window of domains.

When specifying a domain mask from the host---such as when requesting a subset of domains---both formats are accepted. Responses containing domain masks---such as serialized module state---report domain masks in the same format.

If no domain mask is included in the request---implying defaults---serialized state (currently) contains a zero-based mask.

Since domain masks are used as standalone fields, their size is unambiguously determined by the encapsulating wire structure. Therefore, mask structures themselves include no length. Bitmasks returned by the module contain the minimal number of bytes in the actual domain-mask field, skipping trailing zero bytes.

When reported by the module, domain masks are limited to existing domains---i.e., they never describe domains beyond that supported by the originating module.

When importing domain masks, such as when a set of domains is selected for export, any non-existent domains referenced by the mask are ignored. In such cases, the domain mask reported in the response indicates which domains have been processed. This allows trivial detection of a request describing no domains, since the reported domain mask will be ‘‘empty’’, describing a mask with no active domains even if its encoding is not empty of bits.

6.15.1. Zero-based domain mask

Zero-based domain masks contain a bitfield, enumerating domains present as a zero-indexed bitmask.

	offset	field	bytes	notes
1	0	domain mask (field) version	4	reserved 0
2	4	domain mask	<implicit>	bitmask of domains length derived from context

The mask is filled from the left, bit-wise, i.e., a mask describing only domain #1 contains "40..." hex.

6.15.2 Domain mask window

Domain-mask windows contain a bitfield, enumerating domains present starting at a specific domain index.

	offset	field	bytes	notes
1	0	domain mask (field) version	4	reserved 1
2	4	index of first included domain	4	raw integer (zero-based)
3	8	domain mask	<implicit>	bitmask of domains length derived from context

As with zero-based domain masks, bits are filled from the left, bit-wise (6.15.1). The first bit corresponds to the value in the index field: and index field of x0000’0020 with a domain mask of "c0" hex shows domains number 32 and 33 (decimal) being present.

There are no limits on the index of the first domain, such as being a multiple of a power-of-two greater than one. The module may report mask windows with some conventions to simplify reporting---such as, round windows to multiples of 32-bit ‘‘words’’---but incoming indexes are accepted without such requirements.

6.16. Retained key identifier (RKID)

(Semi-)retained key handles are opaque blobs of 64 bytes, (XCP_RETAINID_BYTES). They replace blobs for (semi-)retained keys.

RKIDs combine the key object and its controlling session through a cryptographic hash function. Different instances---blobs---containing the same object will generate different RKIDs.

6.16.1. Truncated Retained key identifier

Administrative management---listing and removal---of RKIDs uses a truncated form of the ID, including only the leading 4 bytes (XCP_RETAINID_SHORT_BYTES).

Note that truncated RKIDs are not guaranteed unique, which only affects administrative uses of RKIDs. Removal affects all RKs with the given truncated ID. We intentionally tolerate collisions during administrative use.

6.17. File-parts and their identification

When transporting files, parts of the file are transported separately. The following ‘‘file-part header’’ identifies individual parts:

	offset		bytes	note
	---	file/part header		
1.	0	file identifier	4	first byte is 0 for all currently supported files, see below
2.	4	start offset	4	
3.	8	file-part bytecount	4	net length, excluding header
	---	file/parts where data is present:		
4.	12	(data)	(var.)	if any, see note

All integer fields are raw integers; see (8.1.3) for a list of recognized identifiers. Note that some of the identifiers, as described there, may be conditionally available, and may be reported as unsupported when referenced in a module lacking them. (There is a separate reason code for out-of-range and conditionally un/supported files, if the distinction is needed for exact error reporting.)

File identifiers are selected with their most significant 8 bits remaining zero, implying 0 in the first byte of this structure. If an alternate structure needs to be added in the future, superseding this one, the first byte will signal further versioning. (Obviously, we do not consider running out of the available 16 million file identifiers possible in 24 bits.)

When querying file contents or length---unambiguous from context---only the file-part header must be present, and no data. When reporting data bytes---during export---or supplying them---during import---bytes must follow the header without any other formatting or padding.

File parts may legitimately contain zero bytes, and no data. (Addressing beyond file limits is reported as a targeting error, but transferring zero bytes in an otherwise valid location is tolerated.) A zero-byte request doubles as size query (see below).

Import and export interfaces dealing with file parts are incremental, allowing targeting parts in arbitrary order.

6.17.1. File-size query

When requesting 0 bytes from offset 0, only a 12-byte header is returned. The file-part bytecount in the header is set to the total file length.

6.17.2. Setting file size

When setting file size---the first call in a file-write sequence---only the file-part header is sent (6.17). In this form, the start offset must be 0, and the bytecount field contains the exact size of the full, reassembled file.

The backend may reject a file-size reservation if the size is over a backend-specific limit.

6.17.3. Removing files

When removing a specific file, a file-part header selects the file to remove (6.17). In this form, offset and bytes fields must still be present, but are currently ignored.

6.18. Bitmasks

Fields encoding bitmasks number their bits as a ‘‘big-endian bitstream’’. The first bit is 0x80 of the first byte. The second bit is 0x40 of the first byte etc. Bitmasks lack size designation or other framing, and rely on range provided by the compound structure they are part of.

Each particular bitmask may specify the index of its first bit; generally, unless marked otherwise, bit #0 is the base index.

7. Other standard formats

We reference a number of industry-standard formats, with some restrictions on which of their possible variants we accept.

7.1. SignerInfo

Digital signatures are encoded into standard ASN.1 signerInfo fields, self-contained: we use SKI-identified signatures, therefore each SignerInfo contains the following fields:

```
SignerInfo ::= SEQUENCE {
        version CMSVersion,
        sid subjectKeyIdentifier [0] SubjectKeyIdentifier,
        digestAlgorithm DigestAlgorithmIdentifier,
        signatureAlgorithm SignatureAlgorithmIdentifier,
        signature SignatureValue
}
```

See SignerInfo in CMS, RFC 3852, Section 5.3. We only use SignerInfo's identified by signer SKI (version 3). The ECDSA format of SignerInfo signature fields is in RFC 5753, Section 7.2.

Currently, the SignerInfo's must not contain authenticated or unauthenticated attributes (both are optional fields of a SignerInfo structure).

7.2. RecipientInfo

See RecipientInfo in CMS, RFC 3852, Section 6.2 (RSA) and 1-pass ECDH version in EC+CMS, RFC 5753, Section 3.1.1 (EC). We identify recipients only through SKI (version 2).

The specific RSA form is KeyTransRecipientInfo, version 2, with rid (recipient ID) containing the recipient's SKI.

Note that we only identify recipients through SKIs, never by certificate issuer/serial number.

7.2.1. EC RecipientInfo

EC recipientInfos supporting 1-pass ECDH must include user key material (UKM) in all forms. Key derivation is currently limited to SHA-256 as a KDF (OID 1.3.132.1.11.1, stdDH-sha256kdf), with AES-256 key wrapping (OID 2.16.840.1.101.3.4.1.45, id-aes256-wrap).

EC points must be always uncompressed.

7.3. SubjectKeyIdentifier (SKI)

See Section 4.2.1.2. in RFC 3280. We derive the SKI from the public key by hashing the BIT STRING subjectPublicKey. The hash function is SHA-256, and full hashes are used, without truncation (32 bytes).

Note that SKIs are tied to a public key, but not to a specific certificate.

7.3.1. Lists of SKIs

When using multiple SKIs, we use a simple concatenated form, packing SKIs in the original order, without any padding or ASN.1 encapsulation. Since the size of SKIs is fixed, this packed form is unambiguous.

Lists of SKIs with an aggregate bytecount not an integer multiple of SKI bytecount (see 7.3) are invalid, and should not be returned by a properly functioning backend.

7.4. SubjectPublicKeyInfo (SPKI)

See Section 1.2 in RFC 4055 (RSA), and 2.1 in RFC 5480 (EC); the latter references section E.5 of ANSI x9.62 for ASN.1 syntax of EC parameters.

7.4.1. SPKI with MAC

For public keys created in, or imported to XCP, the public key structure consists of an SPKI, its attributes, an internally generated salt, and a MAC of the controlling SPKI.

```
                           bytes                  notes
   -----  MACed:  ---------------------------------------------------------
1  SPKI                    as per ASN1 SEQUENCE
2  WK identifier           16                     XCP_WKID_BYTES, see (6.7.1)
3  controlling session     32                     corresponds to blob session
   identifier                                     identifier (3.1 and 6.2.2)
                                                  see also XCP_WK_BYTES
4  salt                    8                      see XCP_SPKISALT_BYTES
5  mode identification     8                      see (3.1)
6  attribute field         (variable)             see (3.2)
   -----  end of MACed region  ---------------------------------------------
7  MAC                     32                     see XCP_HMAC_BYTES
```

Other than the first field (SPKI), fields are individual OCTET STRINGS without further formatting or padding. The MAC is calculated on the all preceding fields by the controlling WKID.

The session identifier matches that of the corresponding private-key blob. It is not used to decrypt, obviously (the SPKI is in the clear) but it may be used to restrict use of the public key for wrapping other keys. It is ignored for encryption and signature verification, which only use the SPKI, and ignore other fields.

Mode identification mimics that of encrypted blobs (3.1). Note that this information is actually redundant: it is also available within the attributes' field. It is replicated for simplified host processing.

The leading SPKI is not prefixed by any data, therefore its size may be derived from its ASN1 length field. If host code parses only the length of first BER construct, it will identify exactly the SPKI, and will ignore subsequent fields. (If a MACed SPKI needs to be embedded in something without explicit control of lengths, we suggest to add a SEQUENCE tag covering it, which will then be correctly parseable.)

7.4.2. SPKIs Montgomery and Edwards curves

With our implementation predating RFC-standardized SPKI formats for relevant non-Weierstrass curves (as of 2018-04-25), we use an ASN.1 encoding extrapolated from prime-field EC SPKIs (2.1 in RFC 5480, E.5 in ANSI x9.62) for Edwards-curve public keys:

```
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ...curve OID...
    }
    BIT STRING
        ...public key...
}
```

Public and private parameter values follow conventions of RFC 7748 and RFC 8032, respectively, are little-endian, with public keys in compressed form. This serialization differs that used by non-Edwards EC coordinates, but the difference is expected to remain opaque to most applications.

See (8.1.1.5.1.) for the list of supported OIDs. Note that we tolerate ‘‘pre-hashed’’ OIDs for ECDH forms, mapping them to the corresponding base forms for both curves.

Note that our SPKI formats are extrapolated from RFC 5480 fields, and differ from some other unofficial definitions (as an example, that used by OpenSSL v1.1.1pre5, 2018-04-17).

7.4.3. PKCS8 private keys for Montgomery and Edwards curves

Similar to EC/Edwards public keys, our implementation predates RFC-standardized PKCS8 formats. We use an ASN.1 encoding extrapolated from prime-field EC private keys [RFC 5915] (SECG SEC1 ECPrivateKey), as PKCS8-encoded structures [RFC 5958, 5].

```
SEQUENCE {
    INTEGER 0                   -- version, 0
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ...curve OID...
    }
    OCTET STRING, encapsulates {
        SEQUENCE {
            INTEGER 1            --  ecdpVer1 (v1), [RFC 5915, Section 3]
            OCTET STRING
                ...private key...
            [1] {
                BIT STRING
                    ...public key...
            }
        }
    }
}
```

Public and private parameter values follow conventions of RFC 7748 and RFC 8032, respectively, are little-endian, with public keys in compressed form.

Inheriting the restriction from [PKCS#11 v2.40 Current Mechanisms Specification, section 2.5], these EC PKCS#8 structures MUST NOT contain the optional ‘‘parameters’’ field.

See (8.1.1.5.1.) for the list of supported OIDs.

7.4.3.1. Sample structures for Montgomery and Edwards curves

Sample keys from RFC 7748 Sections 6.1 and 6.2, and RFC 8032 Sections 7.1 and 7.4 are shown here DER-encoded:

```
-- RFC 7748, Section 6.1, curve25519/x25519(ECDH), Alice public key/SPKI:
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER curveX25519 (1 3 101 110)
    }
    BIT STRING
        85 20 F0 09 89 30 A7 54 74 8B 7D DC B4 3E F7 5A
        0D BF 3A 0D 26 38 1A F4 EB A4 A9 8E AA 9B 4E 6A
```

```
}

-- binary, 53 bytes:
3033300e06072a8648ce3d020106032b656e0321008520f0098930a754748b7d
dcb43ef75a0dbf3a0d26381af4eba4a98eaa9b4e6a

-- RFC 7748, Section 6.1, curve25519/x25519(ECDH), Alice private key:
SEQUENCE {
    INTEGER 0
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER curveX25519 (1 3 101 110)
    }
    OCTET STRING, encapsulates {
        SEQUENCE {
            INTEGER 1
            OCTET STRING
                77 07 6D 0A 73 18 A5 7D 3C 16 C1 72 51 B2 66 45
                DF 4C 2F 87 EB C0 99 2A B1 77 FB A5 1D B9 2C 2A
            [1] {
                BIT STRING
                    85 20 F0 09 89 30 A7 54 74 8B 7D DC B4 3E F7 5A
                    0D BF 3A 0D 26 38 1A F4 EB A4 A9 8E AA 9B 4E 6A
            }
        }
    }
}

-- binary, 99 bytes:
3061020100300e06072a8648ce3d020106032b656e044c304a02010104207707
6d0a7318a57d3c16c17251b26645df4c2f87ebc0992ab177fba51db92c2aa123
0321008520f0098930a754748b7ddcb43ef75a0dbf3a0d26381af4eba4a98eaa
9b4e6a

-- RFC 7748, Section 6.2, x448(ECDH), Alice public key/SPKI:
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER curveX448 (1 3 101 111)
    }
    BIT STRING
        9B 08 F7 CC 31 B7 E3 E6 7D 22 D5 AE A1 21 07 4A
        27 3B D2 B8 3D E0 9C 63 FA A7 3D 2C 22 C5 D9 BB
        C8 36 64 72 41 D9 53 D4 0C 5B 12 DA 88 12 0D 53
        17 7F 80 E5 32 C4 1F A0
}

-- binary, 77 bytes:
304b300e06072a8648ce3d020106032b656f0339009b08f7cc31b7e3e67d22d5
aea121074a273bd2b83de09c63faa73d2c22c5d9bbc836647241d953d40c5b12
da88120d53177f80e532c41fa0

-- RFC 7748, Section 6.2, x448(ECDH), Alice private key:
SEQUENCE {
    INTEGER 0
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER curveX448 (1 3 101 111)
    }
    OCTET STRING, encapsulates {
        SEQUENCE {
            INTEGER 1
            OCTET STRING
                9A 8F 49 25 D1 51 9F 57 75 CF 46 B0 4B 58 00 D4
                EE 9E E8 BA E8 BC 55 65 D4 98 C2 8D D9 C9 BA F5
                74 A9 41 97 44 89 73 91 00 63 82 A6 F1 27 AB 1D
                9A C2 D8 C0 A5 98 72 EB
            [1] {
                BIT STRING
                    9B 08 F7 CC 31 B7 E3 E6 7D 22 D5 AE A1 21 07 4A
                    27 3B D2 B8 3D E0 9C 63 FA A7 3D 2C 22 C5 D9 BB
                    C8 36 64 72 41 D9 53 D4 0C 5B 12 DA 88 12 0D 53
                    17 7F 80 E5 32 C4 1F A0
            }
        }
    }
}

-- binary, 148 bytes:
308191020100300e06072a8648ce3d020106032b656f047c307a02010104389a
8f4925d1519f5775cf46b04b5800d4ee9ee8bae8bc5565d498c28dd9c9baf574
a9419744897391006382a6f127ab1d9ac2d8c0a59872eba13b0339009b08f7cc
31b7e3e67d22d5aea121074a273bd2b83de09c63faa73d2c22c5d9bbc8366472
41d953d40c5b12da88120d53177f80e532c41fa0

-- RFC 8032, Section 7.1, ed25519(EDDSA) KAT public key/SPKI:
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ed25519 (1 3 101 100 1)
    }
    BIT STRING
        D7 5A 98 01 82 B1 0A B7 D5 4B FE D3 C9 64 07 3A
        0E E1 72 F3 DA A6 23 25 AF 02 1A 68 F7 07 51 1A
}

-- binary, 54 bytes:
3034300f06072a8648ce3d020106042b656401032100d75a980182b10ab7d54b
fed3c964073a0ee172f3daa62325af021a68f707511a

-- RFC 8032, Section 7.1, ed25519(EDDSA) KAT private key:
SEQUENCE {
    INTEGER 0
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ed25519 (1 3 101 100 1)
    }
    OCTET STRING, encapsulates {
        SEQUENCE {
            INTEGER 1
            OCTET STRING
                9D 61 B1 9D EF FD 5A 60 BA 84 4A F4 92 EC 2C C4
                44 49 C5 69 7B 32 69 19 70 3B AC 03 1C AE 7F 60
            [1] {
                BIT STRING
                    D7 5A 98 01 82 B1 0A B7 D5 4B FE D3 C9 64 07 3A
                    0E E1 72 F3 DA A6 23 25 AF 02 1A 68 F7 07 51 1A
            }
        }
    }
```

```
}

-- binary, 100 bytes:
3062020100300f06072a8648ce3d020106042b656401044c304a02010104209d
61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60a1
23032100d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68
f707511a

-- RFC 8032, Section 7.4, ed448(EDDSA) KAT public key/SPKI:
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ed448 (1 3 101 100 3)
    }
    BIT STRING
        5F D7 44 9B 59 B4 61 FD 2C E7 87 EC 61 6A D4 6A
        1D A1 34 24 85 A7 0E 1F 8A 0E A7 5D 80 E9 67 78
        ED F1 24 76 9B 46 C7 06 1B D6 78 3D F1 E5 0F 6C
        D1 FA 1A BE AF E8 25 61 80
}

-- binary, 79 bytes:
304d300f06072a8648ce3d020106042b656403033a005fd7449b59b461fd2ce7
87ec616ad46a1da1342485a70e1f8a0ea75d80e96778edf124769b46c7061bd6
783df1e50f6cd1fa1abeafe8256180

-- RFC 8032, Section 7.4, ed448(EDDSA) KAT private key:
SEQUENCE {
    INTEGER 0
    SEQUENCE {
        OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
        OBJECT IDENTIFIER ed448 (1 3 101 100 3)
    }
    OCTET STRING, encapsulates {
        SEQUENCE {
            INTEGER 1
            OCTET STRING
                6C 82 A5 62 CB 80 8D 10 D6 32 BE 89 C8 51 3E BF
                6C 92 9F 34 DD FA 8C 9F 63 C9 96 0E F6 E3 48 A3
                52 8C 8A 3F CC 2F 04 4E 39 A3 FC 5B 94 49 2F 8F
                03 2E 75 49 A2 00 98 F9 5B
            [1] {
                BIT STRING
                    5F D7 44 9B 59 B4 61 FD 2C E7 87 EC 61 6A D4 6A
                    1D A1 34 24 85 A7 0E 1F 8A 0E A7 5D 80 E9 67 78
                    ED F1 24 76 9B 46 C7 06 1B D6 78 3D F1 E5 0F 6C
                    D1 FA 1A BE AF E8 25 61 80
            }
        }
    }
}

-- binary, 151 bytes:
308194020100300f06072a8648ce3d020106042b656403047e307c0201010439
6c82a562cb808d10d632be89c8513ebf6c929f34ddfa8c9f63c9960ef6e348a3
528c8a3fcc2f044e39a3fc5b94492f8f032e7549a20098f95ba13c033a005fd7
449b59b461fd2ce787ec616ad46a1da1342485a70e1f8a0ea75d80e96778edf1
24769b46c7061bd6783df1e50f6cd1fa1abeafe8256180
```

7.5. Certificates

We accept arbitrary x.509 certificates as administrator identities under a reasonable upper limit on size, as long as their public key may be unambiguously decoded. We do not validate attributes of the certificate, or any other parameter other than the public key.

Note that we never verify signatures on certificates. As an important consequence, we never reconstruct certificate hierarchies, and do not need to support revocation.

For certificates of keypart holders---who only encrypt and decrypt, but do not sign data---see (5.3.4).

The upper limit on bytecount, XCP_CERT_MAX_BYTES (1), is only applied by administrative traffic. Additional restrictions, such as those imposed by transport channels, may further restrict the practically usable sizes. The value is also reported as an extended-capability constant.

7.6. PKCS#11 key checksums

We use standard PKCS#11 key checksums where compatible key identification is needed. These checksums are 24-bit ‘‘hashes’’ of the keys; actual calculation depends on key type, as specified by the CKA_CHECK_VALUE attribute. As with standard PKCS#11, collisions are expected and must be tolerated--the only assurance given is that keys with different checksums are different.

7.6.1. Key checksums returned by UnwrapKey

When UnwrapKey() returns a key checksum, the 3-byte PKCS#11 checksum is followed by a 32-bit raw integer, containing the size (bitcount) of the unwrapped key.

7.6.2. Key checksums of PBE mechanisms

Password-based key derivation mechanisms return key checksums concatenated with the IVs derived from the PBE input structure (see 6.8.1.7). The returned value concatenates the 24-bit checksum and the derived IV, the size of which is mechanism-specific.

7.6.3. Public-key types’ checksums

We generalize key checksums to public keys, using the most significant 24 bits of the key’s SKI (7.3). This size matches the size of symmetric CKA_CHECK_VALUE sizes, and conveniently produces the same checksum for a private key and its SPKI.

As with symmetric key checksums, when an SPKI is returned by UnwrapKey, it is followed by a 32-bit raw integer, containing the size (in bits) the unwrapped key.

7.7. ASN.1/BER TLV (tag, length, value) encoding

We use a restricted subset of BER rules to encode payload (see ITU X.680, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation"). We encode requests in the following TLV form:

```
1. all wire packets--request and response packets--are of the form
   "SEQUENCE { OCTET STRING ... }", with the enveloping SEQUENCE
   containing a context-dependent number of OCTET STRINGs.

2. only single-byte tags are supported (hex 30, 04: SEQUENCE, OCTET STRING,
   respectively).

3. length fields are encoded in one of the following forms, depending
   on the bytecount of field contents ('V', the 'Value' field):

  3.1.  up to 127 bytes (0x7f), inclusive, bytecount is inserted as a
        single byte, without any prefix. Therefore, the 4-byte integer
        (hex) 0123'4567 stored into an OCTET STRING appears as
        the following six-byte sequence (bytes are in hex):

             (tag)  (len)      (value)
             04     04         01 23 45 67

  3.2.  up to 255 (0xff) bytes, inclusive, encode length as hex 81,
        followed by a single byte, containing the bytecount itself.
        The integer 0x0123'4567, in this form, would appear as
        the following seven-byte sequence:

             (tag)  (len)      (value)
             04     81 04      01 23 45 67

  3.3.  up to 65'535 (0xffff) bytes, inclusive, encode length as hex 82,
        followed by two bytes, containing the bytecount of the value
        field as a big-endian 16-bite integer. The above integer, in
        this form, would appear as the following eight-byte sequence:

             (tag)  (len)      (value)
             04     82 00 04   01 23 45 67

  3.4.  larger bytecounts extend in a similar way (0x80+N, followed
        by bytecount stored as an N-byte big-endian integer), although
        we do not intend to encounter them in production.

        For practical purposes, host code constructing and understanding
        TL-encodings in the preceding paragraphs would be able to
        communicate with the card without further restrictions.

  3.5.  length encoding does not need to be minimal (i.e., length bytes
        may start with all-zero bytes)

  3.6.  zero-length fields are supported; they must indicate zero
        bytecount, and contain no actual 'V' (Value) bytes.

        A typical example of zero-length fields is optional content,
        where we mandate proper framing--fixed number of OCTET
        STRINGs--with an empty field indicating missing content.

  3.7.  indefinite encoding is never generated by the backend.  It
        must not be present in incoming requests.

Note that zero-byte fields may elicit warnings of ASN.1 parsers,
while it is a valid--arguably, unusual--structure. We acknowledge
this deficiency, and do not intend to fix it: we select simplicity of
the parser vs. eliminating warnings (our use of zero-byte fields is
intentional).
```

7.7.1. Requests

Requests may be encoded with arbitrary-length TL fields. The backend verifies size limits, but otherwise tolerates arbitrary number of Length bytes, as long as they do not exceed the transport limit.

7.7.2. Responses

Responses encode 4-byte fields with single-byte length fields (see item 3.1 under 7.7).

Responses with higher bytecounts are encoded with the minimal number of Length bytes, at least two. Therefore, the backend returns at least the following format:

```
     04'82'<nn>'<mm>     for a bytecount of nn'mm (16-bit hex)
```

In the current form, with current resource limits, all responses over 4 bytes will be encoded in this form, as longer responses are not supported.

7.8. DSA/DH parameters

Structures describing P,Q,G parameters for DSA use the ASN.1/BER encoding specified in RFC 3370 (CMS), section 3.1. The structure used is identified with the OID 1.2.840.10040.4.1 (id-dsa).

Structures describing P and G parameters for DH use the ASN.1/BER encoding specified in PKCS#3, section 9. The structure used is identified with the OID 1.2.840.113549.1.3.1 (dhKeyAgreement).

7.8.1. Raw DSA P,Q,G parameters

As an alternative form, we support concatenated P,Q,G values without ASN.1/BER encapsulation. In this form, all parameters must be stored in the same size (i.e., determined by the size of P), in big-endian form, with leading zeroes used to pad to uniform size. The structure contains P,Q, and G in this order.

7.8.2. Raw DH P,G parameters

As an alternative form, we support concatenated P and G values without ASN.1/BER encapsulation. In this form, all parameters must be stored in the same size (i.e., determined by the size of P), in big-endian form, with leading zeroes used to pad to uniform size. The structure contains P and G in this order.

7.9. Private-key objects

Our blob-internal storage of private-key objects is only documented for interoperable forms, when transporting (un/wrapping) PKCS8-encoded private keys. (The internals of private-key objects are intentionally not standardized, see 3.1.)

For RSA, DSA, DH or Weierstrass-curve EC keys, PKCS#8 PrivateKeyInfo ASN.1 types are used (12.6 in PKCS#11 v2.20, 2.5 in Current Mechanisms Specification v2.40).

See (7.4.3) for notes on Edwards-curve private-key serialization.

8. Constants

This section is generated, extracted from header files. Lists may be appended to, but are not changed, once published.

8.1. Administrative identifiers

See also XCP_ADM_QUERY (0x00010000).

```
---  administrative commands  ----------------------------------------------
XCP_ADM_ADMIN_LOGIN           1 -- add admin certificate
XCP_ADM_DOM_ADMIN_LOGIN       2 -- add domain admin certificate
XCP_ADM_ADMIN_LOGOUT          3 -- revoke admin certificate
XCP_ADM_DOM_ADMIN_LOGOUT.... 4 -- revoke domain admin certificate
XCP_ADM_ADMIN_REPLACE         5 -- transition admin certificate
XCP_ADM_DOM_ADMIN_REPLACE     6 -- transition domain admin certificate
XCP_ADM_SET_ATTR              7 -- set card attribute/s
XCP_ADM_DOM_SET_ATTR........ 8 -- set domain attribute/s
XCP_ADM_GEN_DOM_IMPORTER      9 -- generate new domain importer (PK) key
XCP_ADM_GEN_WK               10 -- create random domain WK
XCP_ADM_EXPORT_WK            11 -- wrap+output WK or parts
XCP_ADM_IMPORT_WK...........12 -- set (set of) WK (parts) to pending
XCP_ADM_COMMIT_WK            13 -- activate pending WK
XCP_ADM_FINALIZE_WK          14 -- promote next to current WK
XCP_ADM_ZEROIZE              15 -- release CSPs from entire module
XCP_ADM_DOM_ZEROIZE.........16 -- release CSPs from domain/s
XCP_ADM_DOM_CTRLPOINT_SET    17 -- fix domain control points
XCP_ADM_DOM_CTRLPOINT_ADD    18 -- enable domain control points
XCP_ADM_DOM_CTRLPOINT_DEL    19 -- disable domain control points
XCP_ADM_SET_CLOCK...........20 -- set module-internal UTC time
XCP_ADM_SET_FCV              21 -- set function-control vector
XCP_ADM_CTRLPOINT_SET        22 -- fix card control points
XCP_ADM_CTRLPOINT_ADD        23 -- enable card control points
XCP_ADM_CTRLPOINT_DEL.......24 -- disable card control points
XCP_ADM_REENCRYPT            25 -- transform blobs to next WK
XCP_ADM_RK_REMOVE            26 -- remove (semi-) retained key
XCP_ADM_CLEAR_WK             27 -- erase current WK
XCP_ADM_CLEAR_NEXT_WK.......28 -- erase pending WK
XCP_ADM_SYSTEM_ZEROIZE       29 -- card zeroize, preserving system
                                -- key, if it is present
XCP_ADM_EXPORT_STATE         30 -- create card state backup
XCP_ADM_IMPORT_STATE         31 -- import card state backup (part)
XCP_ADM_COMMIT_STATE........32 -- activate imported card state
XCP_ADM_REMOVE_STATE         33 -- purge import/export state backup
XCP_ADM_GEN_MODULE_IMPORTER 34 -- generate module importer (PK) key
XCP_ADM_SET_TRUSTED          35 -- activate TRUSTED attribute on
                                -- blob/SPKI
XCP_ADM_DOMAINS_ZEROIZE      36 -- multi-domain zeroize

---  administrative queries  -----------------------------------------------
XCP_ADMQ_ADMIN               0x00010001  -- admin SKI/cert
XCP_ADMQ_DOMADMIN            0x00010002  -- domain admin SKI/cert
XCP_ADMQ_DEVICE_CERT         0x00010003  -- module CA (OA) cert
XCP_ADMQ_DOM_IMPORTER_CERT..0x00010004  -- current WK importer
XCP_ADMQ_CTRLPOINTS          0x00010005  -- card CP set
XCP_ADMQ_DOM_CTRLPOINTS      0x00010006  -- domain CP set
XCP_ADMQ_WK                  0x00010007  -- current WK (domain only)
XCP_ADMQ_NEXT_WK............0x00010008  -- pending WK (domain only)
```

```
XCP_ADMQ_ATTRS              0x00010009  -- card attributes
XCP_ADMQ_DOM_ATTRS          0x0001000a  -- domain attributes
XCP_ADMQ_FCV                0x0001000b  -- public parts of FCV
XCP_ADMQ_WK_ORIGINS.........0x0001000c  -- information on original WK
                                -- components (individual keypart
                                -- verification patterns)
XCP_ADMQ_RKLIST             0x0001000d  -- retained-key id list
XCP_ADMQ_INTERNAL_STATE     0x0001000e  -- (parts of) import/export state
XCP_ADMQ_IMPORTER_CERT      0x0001000f  -- current migration importer
XCP_ADMQ_AUDIT_STATE        0x00010010  -- (parts of) import/export state

8.1.1.  Administrative variables (attributes)

Integer-type variables are referenced through the following indices:

XCP_ADMINT_SIGN_THR        1  -- signature threshold
XCP_ADMINT_REVOKE_THR      2  -- revocation threshold
XCP_ADMINT_PERMS           3  -- permissions
XCP_ADMINT_MODE..........  4  -- operational mode
XCP_ADMINT_STD             5  -- (security) standards compliance

Note that index 0 is not used.

8.1.1.1.  Permissions

XCP_ADMP_WK_IMPORT       0x00000001  -- allow WK import
XCP_ADMP_WK_EXPORT       0x00000002  -- allow WK export
XCP_ADMP_WK_1PART        0x00000004  -- allow transport
                                     -- without multi-part keyparts
XCP_ADMP_WK_RANDOM.......0x00000008  -- allow internally generated WK
XCP_ADMP_1SIGN           0x00000010  -- allow single-signed administration
XCP_ADMP_CP_1SIGN        0x00000020  -- allow single-signed CP modification
XCP_ADMP_ZERO_1SIGN      0x00000040  -- allow single-signed zeroize
XCP_ADMP_NO_DOMAIN_IMPRINT..0x00000080  -- prohibit logging in to domains in
                                     -- imprint mode (card only)
XCP_ADMP_STATE_IMPORT    0x00000100  -- allow state (part) import
XCP_ADMP_STATE_EXPORT    0x00000200  -- allow state (part) export
XCP_ADMP_STATE_1PART     0x00000400  -- allow state transport with 1-part
reserved.................0x00000800  -- reserved for future use
reserved                 0x00001000  -- reserved for future use
XCP_ADMP_CHG_WK_IMPORT   0x00010000  -- allow changing WK import flag
XCP_ADMP_CHG_WK_EXPORT   0x00020000  -- allow changing WK export flag
XCP_ADMP_CHG_WK_1PART....0x00040000  -- allow changing WK transport flag
XCP_ADMP_CHG_WK_RANDOM   0x00080000  -- allow changing internal WK flag
XCP_ADMP_CHG_SIGN_THR    0x00100000  -- allow changing sign threshold
XCP_ADMP_CHG_REVOKE_THR  0x00200000  -- allow changing revoke threshold
XCP_ADMP_CHG_1SIGN.......0x00400000  -- allow changing single-sign
                                     -- threshold setting
XCP_ADMP_CHG_CP_1SIGN    0x00800000  -- allow changing single-sign
                                     -- CP-changing setting
XCP_ADMP_CHG_ZERO_1SIGN  0x01000000  -- allow changing single-sign
                                     -- zeroization setting
XCP_ADMP_CHG_ST_IMPORT   0x02000000  -- allow changing state import bit
                                     -- (ignored by domains)
XCP_ADMP_CHG_ST_EXPORT...0x04000000  -- allow changing state export bit
                                     -- (ignored by domains)
XCP_ADMP_CHG_ST_1PART    0x08000000  -- allow changing 1-part encrypt bit
                                     -- (ignored by domains)
reserved                 0x20000000  -- reserved for future use
reserved                 0x40000000  -- reserved for future use
```

```
WK and CP-related permissions, and those marked as domain-only use are
ignored in card-level attributes. They are rejected if they are supplied
in a card-level set command, and are reported as zeroes when queried.

Permissions controlling state export or import have no domain-level
meaning. These, and attributes marked as non-domain are ignored, but not
modified by the module, when set in domains.

8.1.1.1.1.  Non-modifiable permissions

The following permissions, once removed, may not be added back:

XCP_ADMP_CHG_WK_IMPORT       (0x00010000)
XCP_ADMP_CHG_WK_EXPORT       (0x00020000)
XCP_ADMP_CHG_WK_1PART        (0x00040000)
XCP_ADMP_CHG_WK_RANDOM.......(0x00080000)
XCP_ADMP_CHG_SIGN_THR        (0x00100000)
XCP_ADMP_CHG_REVOKE_THR      (0x00200000)
XCP_ADMP_CHG_1SIGN           (0x00400000)
XCP_ADMP_CHG_CP_1SIGN........(0x00800000)
XCP_ADMP_CHG_ZERO_1SIGN      (0x01000000)
XCP_ADMP_CHG_ST_IMPORT       (0x02000000)
XCP_ADMP_CHG_ST_EXPORT       (0x04000000)
XCP_ADMP_CHG_ST_1PART........(0x08000000)
reserved                     (0x20000000)
reserved                     (0x40000000)

8.1.1.2.  Infrastructure mode

XCP_ADMM_AUTHENTICATED   0x00000001 -- no longer in imprint mode
XCP_ADMM_EXTWNG          0x00000002 -- zeroize if starting with external warning

XCP_ADMM_STR_112BIT      0x00000004 -- require 112+ bits' admin strength
XCP_ADMM_STR_128BIT......0x00000008 -- require 128+ bits' admin strength
XCP_ADMM_STR_160BIT      0x00000010 -- require 160+ bits' admin strength
XCP_ADMM_STR_192BIT      0x00000020 -- require 192+ bits' admin strength
XCP_ADMM_STR_256BIT      0x00000040 -- require 256  bits' admin strength

XCP_ADMM_WKCLEAN_EXTWNG..0x00000080 -- zeroize WKs if starting with ext. warning
                                    -- set.  Leaves other parameters unaffected
XCP_ADMM_BATT_LOW        0x00000100 -- module reports low battery (read only)
XCP_ADMM_API_ACTIVE      0x00000200 -- XCP commands are available.  Remove bit
                                    -- to disable XCP within card, such as
                                    -- during card movement.

Bits related to the external warning infrastructure or global state
(XCP_ADMM_EXTWNG, XCP_ADMM_BATT_LOW, XCP_ADMM_WKCLEAN_EXTWNG,
XCP_ADMM_API_ACTIVE) within domain attributes are read-only; they are
mirrored from the card-level one. These read-only bits are ignored when
attempting a write at the domain level.

8.1.1.3.  Operational mode (compliance settings)

Compliance settings correspond to standards-mandated sets of CPs. They
are read-only, and are updated when CPs are updated, or a domain changes
state.

XCP_ADMS_FIPS2009       1  -- NIST, 80+ bits,  -2011.01.01.
XCP_ADMS_BSI2009        2  -- BSI, 80+ bits,  -2011.01.01.
```

```
XCP_ADMS_FIPS2011       4  -- NIST, 112+ bits,  2011.01.01.-
XCP_ADMS_BSI2011        8  -- BSI,  112+ bits,  2011.01.01.-

The following bits are reserved:
XCP_ADMS_SIGG_IMPORT   16  -- German SigG (digital signature law),
                           -- mode allows key import but not export
XCP_ADMS_SIGG          32  -- German SigG, no key import

XCP_ADMS_BSICC2017     64  -- BSI, EP11 EAL4 Common Criteria
                           -- Certification 2017
```

When supplied to the backend as a requested compliance restriction, these values are encoded as a 32-bit bitfield of the CKA_IBM_STD_COMPLIANCE1 integer attribute, see (6.8.3.1) for usage.

8.1.1.3.1. FIPS/2009 mode

This mode, corresponding to FIPS-140 restrictions in 2009, is active if the following CPs are prohibited:

```
XCP_CPB_KEYSZ_BELOW80BIT
XCP_CPB_ALG_RAW_RSA
XCP_CPB_SKIP_KEYTESTS
XCP_CPB_ALG_NFIPS2009
XCP_CPB_KEYSZ_HMAC_ANY
```

Note that the presence of BP (non-NIST) EC curves no longer prohibits FIPS mode. This is due to a policy change, not a technical one.

8.1.1.3.2. FIPS/2011 mode

This mode, corresponding to FIPS-140 restrictions on 2011.01.01 (transition off 80-bit strength, and transition to FIPS 186-3), is active if the following CPs are prohibited:

```
XCP_CPB_KEYSZ_BELOW80BIT
XCP_CPB_KEYSZ_80BIT           -- new addition since FIPS/2009
XCP_CPB_ALG_RAW_RSA
XCP_CPB_SKIP_KEYTESTS
XCP_CPB_ALG_NFIPS2011         -- replaces algorithm choice since FIPS/2009
XCP_CPB_KEYSZ_HMAC_ANY
XCP_CPB_KEYSZ_RSA65536        -- new addition since FIPS/2009 (FIPS 186-3)
```

Note that the presence of BP (non-NIST) EC curves no longer prohibits FIPS mode. This is due to a policy change, not a technical one.

8.1.1.3.3. BSI/2009 mode

This mode, corresponding to the BSI HSM protection profile, and German Bundesnetzagentur algorithms in 2009, is active if the following CPs are prohibited:

```
XCP_CPB_KEYSZ_BELOW80BIT
XCP_CPB_NON_ATTRBOUND
XCP_CPB_ALG_RAW_RSA
XCP_CPB_SKIP_KEYTESTS
XCP_CPB_ALG_NBSI2009
XCP_CPB_KEYSZ_HMAC_ANY
```

```
8.1.1.3.4.  BSI/2011 mode

This mode, corresponding to the BSI HSM protection profile, and German
Bundesnetzagentur algorithms as of 2011.01.01., is active if the
following CPs are prohibited:

XCP_CPB_KEYSZ_BELOW80BIT
XCP_CPB_KEYSZ_80BIT           -- new addition since BSI/2009
XCP_CPB_NON_ATTRBOUND
XCP_CPB_ALG_RAW_RSA
XCP_CPB_SKIP_KEYTESTS
XCP_CPB_ALG_NBSI2011         -- replaces algorithm choice since BSI/2009
XCP_CPB_KEYSZ_HMAC_ANY

8.1.1.3.5.  Common Criteria Certification mode

This mode, corresponding to the CP settings evaluated for Common Criteria
Certification by German Bundesnetzagentur, is active if the following CP
settings are applied:

XCP_CPB_ADD_CPBS            ( 0)    n/r
XCP_CPB_DELETE_CPBS         ( 1)    n/r
XCP_CPB_SIGN_ASYMM          ( 2)     1
XCP_CPB_SIGN_SYMM...........( 3)... n/r
XCP_CPB_SIGVERIFY_SYMM      ( 4)    n/r
XCP_CPB_ENCRYPT_SYMM        ( 5)     1
XCP_CPB_DECRYPT_ASYMM       ( 6)     1
XCP_CPB_DECRYPT_SYMM........( 7)...  1
XCP_CPB_WRAP_ASYMM          ( 8)     1
XCP_CPB_WRAP_SYMM           ( 9)     1
XCP_CPB_UNWRAP_ASYMM        (10)     1
XCP_CPB_UNWRAP_SYMM.........(11)...  1
XCP_CPB_KEYGEN_ASYMM        (12)     1
XCP_CPB_KEYGEN_SYMM         (13)     1
XCP_CPB_RETAINKEYS          (14)     1
XCP_CPB_SKIP_KEYTESTS.......(15)...  0
XCP_CPB_NON_ATTRBOUND       (16)     0
XCP_CPB_MODIFY_OBJECTS      (17)     1
XCP_CPB_RNG_SEED            (18)     0
XCP_CPB_ALG_RAW_RSA.........(19)...  0
XCP_CPB_ALG_NFIPS2009       (20)     0
XCP_CPB_ALG_NBSI2009        (21)     1
XCP_CPB_KEYSZ_HMAC_ANY      (22)     0
XCP_CPB_KEYSZ_BELOW80BIT....(23)...  0
XCP_CPB_KEYSZ_80BIT         (24)     1
XCP_CPB_KEYSZ_112BIT        (25)     1
XCP_CPB_KEYSZ_128BIT        (26)     1
XCP_CPB_KEYSZ_192BIT........(27)...  1
XCP_CPB_KEYSZ_256BIT        (28)     1
XCP_CPB_KEYSZ_RSA65536      (29)     0
XCP_CPB_ALG_RSA             (30)     1
XCP_CPB_ALG_DSA.............(31)...  1
XCP_CPB_ALG_EC              (32)     1
XCP_CPB_ALG_EC_BPOOLCRV     (33)     1
XCP_CPB_ALG_EC_NISTCRV      (34)     1
XCP_CPB_ALG_NFIPS2011.......(35)...  1
XCP_CPB_ALG_NBSI2011        (36)     1
XCP_CPB_USER_SET_TRUSTED    (37)     0
XCP_CPB_ALG_SKIP_CROSSCHK   (38)     0
XCP_CPB_WRAP_CRYPT_KEYS.....(39)...  0
```

XCP_CPB_SIGN_CRYPT_KEYS	(40)	0
XCP_CPB_WRAP_SIGN_KEYS	(41)	0
XCP_CPB_USER_SET_ATTRBOUND	(42)	0
XCP_CPB_ALLOW_PASSPHRASE	(43)	0
XCP_CPB_WRAP_STRONGER_KEY	(44)	0
XCP_CPB_WRAP_WITH_RAW_SPKI	(45)	0
XCP_CPB_ALG_DH	(46)	1
XCP_CPB_DERIVE	(47)	1
XCP_CPB_ALG_EC_25519	(55)	n/r
XCP_CPB_ALG_NBSI2017	(61)	n/r
XCP_CPB_CPACF_PK	(64)	n/r
XCP_CPB_ALG_PQC	(65)	n/r

Apart from the technical enforcement of the Common Criteria Certification mode through the control point settings, please note that the Common Criteria Certification only applies to CryptoExpress cards that are used within an IBM zSeries mainframe platform that runs a Common Criteria evaluated version of the z/OS operating system.

8.1.1.4. Importer keytypes

The following importer key types/sizes are currently defined:

XCP_IMPRKEY_RSA_2048	0
XCP_IMPRKEY_RSA_4096	1
XCP_IMPRKEY_EC_P256	2
XCP_IMPRKEY_EC_P521	3
XCP_IMPRKEY_EC_BP256r	4
XCP_IMPRKEY_EC_BP320r	5
XCP_IMPRKEY_EC_BP512r	6
XCP_IMPRKEY_RSA_3072	7

RSA keys are restricted to Fermat exponents (0x10001). EC keys are from NIST or BP/Brainpool (regular) curves.

The available set of importer types is a superset of accepted administrator key types/sizes.

8.1.1.5. EC curve identifiers

Symbolic (non-OID) identification of EC curves, such as used by the audit infrastructure, uses these constants:

XCP_EC_C_NIST_P192	1	-- NIST, prime field
XCP_EC_C_NIST_P224	2	
XCP_EC_C_NIST_P256	3	
XCP_EC_C_NIST_P384	4	
XCP_EC_C_NIST_P521	5	
XCP_EC_C_BP160R	6	-- Brainpool, prime field
XCP_EC_C_BP160T	7	
XCP_EC_C_BP192R	8	
XCP_EC_C_BP192T	9	
XCP_EC_C_BP224R	10	
XCP_EC_C_BP224T	11	
XCP_EC_C_BP256R	12	
XCP_EC_C_BP256T	13	
XCP_EC_C_BP320R	14	
XCP_EC_C_BP320T	15	
XCP_EC_C_BP384R	16	
XCP_EC_C_BP384T	17	

```
XCP_EC_C_BP512R          18
XCP_EC_C_BP512T          19
XCP_EC_C_25519...........20..-- curve/x25519, ECDH only
reserved                 21  -- reserved for future use
reserved                 22  -- reserved for future use
XCP_EC_C_SECP256K1       23  -- secp256k1 (Bitcoin default)
XCP_EC_C_ED448           24  -- ed448 Goldilocks twisted-Edwards
                             -- FP, 2^448-2^224+1
XCP_EC_C_448             25  -- c448/x448, ECDH only
XCP_EC_C_ED25519         26  -- ed25519, EdDSA
```

8.1.1.5.1. Non-Weierstrass EC curve identifiers

The following IETF-standarized (RFC 8410) OIDs are supported for c25519/c448 variants:

ECDH usage

```
1.3.101.110  -- curve25519 with mechanisms CKM_IBM_EC_X25519 and
             -- CKM_IBM_EC_X25519_RAW
1.3.101.111  -- curve448 with mechanisms CKM_IBM_EC_X448 and CKM_IBM_EC_X448_RAW
```

EDDSA usage

```
1.3.101.112  -- ed25519 with mechanism CKM_IBM_ED25519_SHA512
1.3.101.113  -- ed448 with mechanism CKM_IBM_ED448_SHA3
```

8.1.1.6. EC curve-group identifiers

```
XCP_EC_CG_NIST        1  -- NIST, prime field
XCP_EC_CG_BPOOL       2  -- Brainpool, FP curves
                         -- including R and twisted (T) parameters
XCP_EC_CG_C25519      3  -- curve25519 (RFC 7748)
XCP_EC_CG_SECP256K1.. 4..-- SECP K-curves, incl. Bitcoin default secp256k1
reserved              5  -- reserved for future use
XCP_EC_CG_C448       6  -- ed448 ("Goldilocks") (RFC 7748)
```

8.1.2. Serialized state types (tags)

Type-specific information is in (5.3). This listing enumerates our exported names for the type constants in alphabetical order; see the type listing for use.

```
XCP_STSTYPE_CARD_ADM_CERTS     x0009
XCP_STSTYPE_CARD_ADM_SKIS      x0008
XCP_STSTYPE_CARD_ATTRS         x000e
XCP_STSTYPE_CARD_QUERY.........x0007
XCP_STSTYPE_CARD_TRANSCTR      x0010
XCP_STSTYPE_CERT_AUTH          x001e
XCP_STSTYPE_CPS_MASK           x0021
XCP_STSTYPE_CREATE_TIME........x0005
XCP_STSTYPE_DOMAINIDX_MAX      x0002
XCP_STSTYPE_DOMAINS_MASK       x0003
XCP_STSTYPE_DOM_ADM_CERTS      x000b
XCP_STSTYPE_DOM_ADM_SKIS.......x000a
XCP_STSTYPE_DOM_ATTRS          x000f
XCP_STSTYPE_DOM_CPS            x0017
XCP_STSTYPE_DOM_QUERY          x000c
XCP_STSTYPE_DOM_TRANSCTR.......x0011
```

XCP_STSTYPE_FCV	x0006
XCP_STSTYPE_FILE_SIG	x0016
XCP_STSTYPE_KEYPART	x0019
XCP_STSTYPE_KEYPART_CERT	x001d
XCP_STSTYPE_KEYPART_COUNT	x001b
XCP_STSTYPE_KEYPART_LIMIT	x001c
XCP_STSTYPE_KEYPART_SIG	x001a
XCP_STSTYPE_KPH_SKIS	x000d
XCP_STSTYPE_MULTIIMPORT_MASK	x0020
XCP_STSTYPE_SECTIONCOUNT	x0001
XCP_STSTYPE_SERIALNR	x0004
XCP_STSTYPE_SIG_CERTS	x0015
XCP_STSTYPE_SIG_CERT_COUNT	x0014
XCP_STSTYPE_STATE_SALT	x0018
XCP_STSTYPE_STATE_SCOPE	x001f
XCP_STSTYPE_WK_ENCR_ALG	x0012
XCP_STSTYPE_WK_ENCR_DATA	x0013

8.1.3. File identifiers

When a conditionally-present file identifier is referenced in a module not supporting it, it is reported as recognized but unsupported.

XCP_FILEID_SAVED_STATE	1	
XCP_FILEID_KEYPARTS	2	
XCP_FILEID_TESTDATA	3	-- not present in production firmware
XCP_FILEID_EXPREQUEST	4	-- test file containing export-request -- parameters (keyparts, thresholds etc.)

Test-data files, if supported, are used only by development features, and not further specified here. Please consult any internal documentation of the test feature itself.

8.2. Function identifiers

FNID_Login	1
FNID_Logout	2
FNID_SeedRandom	3
FNID_GenerateRandom	4
FNID_DigestInit	5
FNID_DigestUpdate	6
FNID_DigestFinal	8
FNID_Digest	9
FNID_DigestSingle	10
FNID_EncryptInit	11
FNID_DecryptInit	12
FNID_EncryptUpdate	13
FNID_DecryptUpdate	14
FNID_EncryptFinal	15
FNID_DecryptFinal	16
FNID_Encrypt	17
FNID_Decrypt	18
FNID_EncryptSingle	19
FNID_DecryptSingle	20
FNID_GenerateKey	21
FNID_GenerateKeyPair	22
FNID_SignInit	23
FNID_SignUpdate	24
FNID_SignFinal	25
FNID_Sign	26

```
FNID_VerifyInit           27
FNID_VerifyUpdate.......... 28
FNID_VerifyFinal          29
FNID_Verify               30
FNID_SignSingle           31
FNID_VerifySingle.......... 32
FNID_WrapKey              33
FNID_UnwrapKey            34
FNID_DeriveKey            35
FNID_GetMechanismList...... 36
FNID_GetMechanismInfo     37
FNID_get_xcp_info         38
FNID_GetAttributeValue    39
FNID_SetAttributeValue..... 40
FNID_admin                41

Functions not supported:
FNID_DigestKey             7
FNID_ReencryptSingle      42


Note that all function identifiers are non-zero.

8.2.1.  Function (performance) category identifiers

The returned transport header includes a performance categorization
of functional requests. Requests are categorized into two performance
classes with symmetric requests (incremental vs. single-pass
symmetric operations), fast/slow asymmetric operations. A separate
category reserved for key generation including primality testing, or
similarly slow requests.

The internal classification of fast vs. slow, especially for asymmetric
operations, is implementation-dependent, such as the performance of
hardware acceleration. The classification may be assumed to be stable
when identical firmware versions are loaded into identical hardware.

XCP_OPCAT_ASYMM_SLOW      1
XCP_OPCAT_ASYMM_FAST      2  -- includes EC key generation
XCP_OPCAT_SYMM_PARTIAL    3  -- includes hashing
XCP_OPCAT_SYMM_FULL...... 4  -- includes symmetric key generation
XCP_OPCAT_ASYMM_GEN       5  -- RSA, DSA parameter generation
XCP_OPCAT_ASYMM_MAX       5  -- total number of categories

8.3.  Card control points

Card control points are currently not supported.  Command codes
are reserved for their manipulation, but there are no card-level
CP definitions.

8.4.  Domain control points (CPs)

Control points are positive-active, requiring a nonzero bit to activate
a capability (CPB, CP bit). Their full set of CPBs is represented as a
XCP_CPCOUNT-bit wide fixed field, a bitmask (6.18).

The list of control points may be appended to, but existing CP bits'
meanings are not changed, once released. Possible gaps in the list are
reserved for future use and can not be activated. The reported maximum
control point bit might be greater than the last documented bit, too.
```

```
XCP_CPB_ADD_CPBS            0 -- allow addition (activation) of CP bits
XCP_CPB_DELETE_CPBS         1 -- allow removal (deactivation) of CP bits
                              -- remove both ADD_CPBs and DELETE_CPBs
                              -- to make unit read-only
XCP_CPB_SIGN_ASYMM          2 -- sign with private keys
XCP_CPB_SIGN_SYMM........... 3.-- sign with HMAC or CMAC
XCP_CPB_SIGVERIFY_SYMM      4 -- verify with HMAC or CMAC
XCP_CPB_ENCRYPT_SYMM        5 -- encrypt with symmetric keys
                              -- No asymmetric counterpart: one
                              -- may not restrict use of public keys
XCP_CPB_DECRYPT_ASYMM       6 -- decrypt with private keys
XCP_CPB_DECRYPT_SYMM........ 7.-- decrypt with symmetric keys
XCP_CPB_WRAP_ASYMM          8 -- key export with public keys
XCP_CPB_WRAP_SYMM           9 -- key export with symmetric keys
XCP_CPB_UNWRAP_ASYMM       10 -- key import with private keys
XCP_CPB_UNWRAP_SYMM.........11.-- key import with symmetric keys
XCP_CPB_KEYGEN_ASYMM       12 -- generate asymmetric keypairs
XCP_CPB_KEYGEN_SYMM        13 -- generate or derive symmetric keys
                              -- including DSA parameters
XCP_CPB_RETAINKEYS         14 -- allow backend to save semi/retained keys
XCP_CPB_SKIP_KEYTESTS.......15.-- disable selftests on new asymmetric keys
XCP_CPB_NON_ATTRBOUND      16 -- allow keywrap without attribute-binding
XCP_CPB_MODIFY_OBJECTS     17 -- allow changes to objects (Booleans only)
XCP_CPB_RNG_SEED           18 -- allow mixing external seed to RNG
XCP_CPB_ALG_RAW_RSA.........19.-- allow RSA private-key use without padding
                              -- (highly discouraged)
XCP_CPB_ALG_NFIPS2009      20 -- allow non-FIPS-approved algs (as of 2009)
                              -- including non-FIPS keysizes
XCP_CPB_ALG_NBSI2009       21 -- allow non-BSI algorithms (as of 2009)
                              -- including non-FIPS keysizes
XCP_CPB_KEYSZ_HMAC_ANY     22 -- don't enforce minimum keysize on HMAC
XCP_CPB_KEYSZ_BELOW80BIT....23.-- allow algorithms below 80-bit strength
                              -- public-key operations are still allowed
XCP_CPB_KEYSZ_80BIT        24 -- allow 80  to 111-bit algorithms
XCP_CPB_KEYSZ_112BIT       25 -- allow 112 to 127-bit algorithms
XCP_CPB_KEYSZ_128BIT       26 -- allow 128 to 191-bit algorithms
XCP_CPB_KEYSZ_192BIT........27.-- allow 192 to 255-bit algorithms
XCP_CPB_KEYSZ_256BIT       28 -- allow 256-bit        algorithms
XCP_CPB_KEYSZ_RSA65536     29 -- allow RSA public exponents below 0x10001
XCP_CPB_ALG_RSA            30 -- RSA private-key or key-encrypt use
XCP_CPB_ALG_DSA.............31.-- DSA private-key use
XCP_CPB_ALG_EC             32 -- EC private-key use, see also
                              -- curve restrictions
XCP_CPB_ALG_EC_BPOOLCRV    33 -- Brainpool (E.U.) EC curves
XCP_CPB_ALG_EC_NISTCRV     34 -- NIST/SECG EC curves
XCP_CPB_ALG_NFIPS2011.......35.-- allow non-FIPS-approved algs (as of 2011)
                              -- including non-FIPS keysizes
XCP_CPB_ALG_NBSI2011       36 -- allow non-BSI algorithms (as of 2011)
                              -- including non-BSI keysizes
XCP_CPB_USER_SET_TRUSTED   37 -- allow non-admins to set TRUSTED on a blob/SPKI
XCP_CPB_ALG_SKIP_CROSSCHK  38 -- do not double-check sign/decrypt ops
XCP_CPB_WRAP_CRYPT_KEYS.....39.-- allow keys which can en/decrypt data
                              -- and also un/wrap other keys
XCP_CPB_SIGN_CRYPT_KEYS    40 -- allow keys which can en/decrypt data
                              -- and also sign/verify
XCP_CPB_WRAP_SIGN_KEYS     41 -- allow keys which can un/wrap data
                              -- and also sign/verify
XCP_CPB_USER_SET_ATTRBOUND 42 -- allow non-administrators to
                              -- mark public key objects ATTRBOUND
```

```
XCP_CPB_ALLOW_PASSPHRASE....43.-- allow host to pass passprases, such as
                               -- PKCS12 data, in the clear
XCP_CPB_WRAP_STRONGER_KEY   44 -- allow wrapping of stronger keys
                               -- by weaker keys
XCP_CPB_WRAP_WITH_RAW_SPKI  45 -- allow wrapping with SPKIs without
                               -- MAC and attributes
XCP_CPB_ALG_DH              46 -- Diffie-Hellman use (private keys)
XCP_CPB_DERIVE..............47 -- allow key derivation (symmetric+EC/DH)
XCP_CPB_ALG_EC_25519        55 -- enable support of curve25519 and its
                               -- related algorithms incl. EdDSA
XCP_CPB_ALG_NBSI2017        61 -- allow non-BSI algorithms (as of 2017)
                               -- including non-BSI keysizes

XCP_CPB_CPACF_PK            64 -- support data key generation and import
                               -- for protected key
XCP_CPB_ALG_PQC             65 -- enable support for PQC algorithms
```

8.5. Blob Boolean attributes

The following bits are defined, matching PKCS#11 or vendor-defined
attributes (see storage in 3.2.2).

```
XCP_BLOB_EXTRACTABLE               0x00000001
XCP_BLOB_NEVER_EXTRACTABLE         0x00000002
XCP_BLOB_MODIFIABLE                0x00000004
XCP_BLOB_NEVER_MODIFIABLE..........0x00000008
XCP_BLOB_RESTRICTABLE              0x00000010 -- may remove, but not add, capabilities
XCP_BLOB_LOCAL                     0x00000020 -- was locally generated, not imported
XCP_BLOB_ATTRBOUND                 0x00000040 -- must be transported with attributes
XCP_BLOB_USE_AS_DATA...............0x00000080 -- raw bytes may be used as input
                                              -- such as for DeriveKey
XCP_BLOB_SIGN                      0x00000100
XCP_BLOB_SIGN_RECOVER              0x00000200
XCP_BLOB_DECRYPT...................0x00000400
XCP_BLOB_ENCRYPT                   0x00000800
XCP_BLOB_DERIVE                    0x00001000
XCP_BLOB_UNWRAP                    0x00002000
XCP_BLOB_WRAP......................0x00004000
XCP_BLOB_VERIFY                    0x00008000
XCP_BLOB_VERIFY_RECOVER            0x00010000
XCP_BLOB_TRUSTED                   0x00020000
XCP_BLOB_WRAP_W_TRUSTED............0x00040000
XCP_BLOB_RETAINED                  0x00080000 -- blob resides within backend
XCP_BLOB_ALWAYS_RETAINED           0x00100000 -- blob has been created within backend
XCP_BLOB_MLS                       0x00800000 -- blob interacts with MLS functionality
XCP_BLOB_PROTKEY_EXTRACTABLE.......0x00200000 -- blob can be imported as protected key
                                              -- conflicts with XCP_BLOB_EXTRACTABLE and
                                              -- sets XCP_BLOB_NEVER_EXTRACTABLE
XCP_BLOB_PROTKEY_NEVER_EXTRACTABLE 0x00400000
```

8.5.1. Vendor-defined attributes (CKA_IBM_...)

Vendor-defined Boolean attributes are referenced through the following
constants:

```
CKA_IBM_RESTRICTABLE               0x80010001
CKA_IBM_NEVER_MODIFIABLE           0x80010002
CKA_IBM_RETAINKEY                  0x80010003
CKA_IBM_ATTRBOUND..................0x80010004
```

```
CKA_IBM_USE_AS_DATA               0x80010008
CKA_IBM_STD_COMPLIANCE1           0x8001000a
CKA_IBM_MLS_TYPE                  0x8001000b
CKA_IBM_PROTKEY_EXTRACTABLE........0x8001000c
CKA_IBM_PROTKEY_NEVER_EXTRACTABLE  0x8001000d
```

Note that we use the regular constant to indicate
implementation-dependent attributes (CKA_VENDOR_DEFINED, 0x80000000).

The following vendor-defined attribute is also used:

```
CKA_IBM_STRUCT_PARAMS      0x80010009  (6.8.1.6) (7.8) (7.8.1)
```

8.6. Test mechanism constants

The following operation IDs are available for test-only calls:

```
XCP_DEV_SET_WK               1  -- imprints affected domain,
                                -- in addition to setting current WK
XCP_DEV_SET_NEXT_WK          2
XCP_DEV_AES_ENCR_CYCLE       3
XCP_DEV_AES_DECR_CYCLE.....  4
XCP_DEV_DES_ENCR_CYCLE       5
XCP_DEV_DES_DECR_CYCLE       6
XCP_DEV_ZEROIZE_CARD         7
XCP_DEV_ZEROIZE_DOMAIN.....  8
XCP_DEV_SET_DOMAIN_CPS       9
XCP_DEV_SET_WK_RAW          10  -- does not imprint affected domain
XCP_DEV_COMMIT_NEXT_WK      11  -- commits next WK
XCP_DEVQ_ADMINLIST......... 12..-- list of card SKIs
XCP_DEVQ_DOM_ADMINLIST      13  -- list of domain SKIs
XCP_DEV_SET_NEXT_WK_RAW     14  -- set next WK, does not imprint
XCP_DEV_FSMODE              15  -- manage access to filesystems
                                -- used to simulate parts of concurrent-update
XCP_DEV_ADMSIGN............ 16..-- admin-sign file in filesystem
                                -- used to pass arbitrary bad structs
                                -- for verification
XCP_DEV_FSWRITE             17  -- write data to temporary file
XCP_DEV_DSA_PQG_GEN         18  -- turn P, Q bitcount +iteration count plus
                                -- (optional) seed into DSA PQG parameter set
XCP_DEVQ_BLOBCONFIG         19  -- CSP/blob configuration details
                                -- see (5.2.6.) for format details
XCP_DEV_RSA_X931_KEYGEN.... 20..-- ANSI x9.31 key gen. from prime seeds
                                -- returns PKCS8-encoded key, in clear
                                -- may be unsupported, depending on Clic setup
XCP_DEV_RNGSTATE            21  -- query or set backend RNG state
XCP_DEV_RNG_SEED            22  -- forces immediate RNG re/seeding
                                -- recommended before exporting RNG state,
                                -- to maximize number of matching bits after
                                -- state is restored
XCP_DEVQ_ENTROPY            23  -- retrieve raw TRNG output
                                -- conditioned entropy, no DRNG processing
                                -- note that this call changes DRNG setup
                                -- during processing, slowing it down
XCP_DEVQ_PERFMODE.......... 24..-- query performance/timestamp setup
XCP_DEV_PERFMODE            25  -- change performance/timestamp setup
XCP_DEV_RSA_DECR_CYCLE      26  -- RSA, raw modular exponentiation, looped
XCP_DEV_RSACRT_DECR_CYCLE   27  -- RSA, private exponent, CRT, looped
XCP_DEV_ECMUL_CYCLE........ 28..-- EC scalar multiplication, looped
```

```
XCP_DEVQ_PERF_LOCK          30  -- raw performance: un/lock cycles,
                                -- single thread
                                -- test on otherwise quiesced backend
XCP_DEVQ_PERF_WAKE          31  -- raw performance: un/lock cycles,
                                -- forcing context switching
                                -- test on otherwise quiesced backend
XCP_DEVQ_PERF_SCALE         32  -- raw performance: add calibrating
                                -- timestamp/syslog/etc. entries
                                -- to simplify offline scaling of
                                -- performance management

XCP_DEV_CACHE_MODE......... 33..-- set or query module-internal cache state
XCP_DEVQ_CACHE_STATS        34  -- log cache-statistics summary
                                -- over syslog etc. (system-dependent)
XCP_DEV_DELAY               35  -- NOP: delay the backend thread by
                                -- a host-influenced amount of time,
                                -- without performing other operations

                                -- transport/stresstest functions
XCP_DEV_COMPRESS........... 36..-- return 'summarized' version of any
                                -- supplied data
XCP_DEV_XOR_FF              37  -- returns a copy of data, all bits
                                -- flipped (XORed with 0xff)
XCP_DEV_PRF                 38  -- returns PRF stream from caller-
                                -- provided seed and bytecount

XCP_DEV_TRANSPORTSTATE1     39  -- transport-statistics dump
                                -- (system-dependent functionality)

XCP_DEVQ_CACHEINDEX........ 40..-- return module-internal blob index
                                -- of caller-provided key
XCP_DEVQ_CSP_OBJCOUNT       41  -- CSP-object reference counter,
                                -- if available
XCP_DEV_CSPTYPE             42  -- preferred CSP-object (engine) type
XCP_DEV_FCV                 43  -- query and/or set current FCV
                                -- without signature verification
XCP_DEV_CLEAR_FCV.......... 44  -- erase any existing FCV
XCP_DEVQ_ASSERTIONS         45  -- verify the consistency of module-
                                -- internal data structures, as a
                                -- stronger form of assertion-checking
XCP_DEV_TEST_LATESTART      46  -- perform any initial test which has
                                -- been skipped during backend startup.
                                -- Not necessary unless running against
                                -- the most aggressive SYS_TEST_0START
                                -- settings [which trim down backend
                                -- testing to a bare minimum]
XCP_DEV_ENVSEED             47  -- seed backend with device-unique,
                                -- environment-derived, low quality
                                -- entropy [augments/replaces real
                                -- entropy seeding; added to let
                                -- unrelated backends diverge even
                                -- when VM-hosted or otherwise lacking
                                -- proper initial entropy]
XCP_DEVQ_RAWENTROPY........ 48  -- retrieve raw entropy pool, before
                                -- compression, if backend supports
                                -- this. Details are CSP-specific.
                                -- see also: XCP_DEV_ENTROPY
XCP_DEV_EC_SIGVER_CYCLE     49  -- EC sign/verify in loop
                                -- operation derived from supplied blob
                                -- (EC private/SPKI)
                                -- see also: XCP_DEV_ECMUL_CYCLE
XCP_DEV_DRAIN_ENTROPY       50  -- TRNG: force one entropy->compression call
                                -- [may be more, depends on DRNG state]
XCP_DEV_CONV_EC_BLOB        51  -- development-internal conversion function
                                -- details not relevant/published
XCP_DEVQ_COUNTERS.......... 52  -- retrieve coverage counters
                                -- these are intentionally not published
                                -- [development-internal use only, contact us]
XCP_DEV_RSACRT_MSG_CYCLE    53  -- RSA, private exponent, CRT, looped
                                -- this variant increments message
                                -- and returns per-message statistics

XCP_DEV_AUDIT_CYCLE         54  -- audit-log, generate log entries
                                -- in a loop

XCP_DEV_PQC_DILITHIUM       55  -- post-quantum algs: generic call
                                -- for Dilithium (from PQ Crystals)
```

These functions are restricted to development, since many of them
perform insecure operations. Some of this functionality mimics regular
queries, without generating card signatures, therefore considerably
faster if performed frequently.

8.6.1. RNG test constants

During RNG-stream testing, the type of RNG test must be selected. The
following constants are defined:

```
XCP_DEV_RNG_TRNG        0 -- no DRNG involvement
XCP_DEV_RNG_DRNG        1 -- DRNG, no reseeding
XCP_DEV_RNG_MIXED       2 -- DRNG, with TRNG reseeding
```

8.6.2. Blob-cache control

When setting blob state, the following bits may be supplied:

```
XCP_DEVC_CACHE_ACTIVE        1  -- activate blob-cache
XCP_DEVC_CACHE_INACTIVE      2  -- suspend caching: lookups fail,
                                -- new entries are not accepted
XCP_DEVC_CACHE_FLUSH         4  -- evict all currently cached objects
                                -- available even if cache is suspended
```

The returned value contains exactly one of ACTIVE or INACTIVE, and no
other bit.

8.7. Other fixed values

```
MOD_VARLENS_BYTES             8
MOD_WRAP_BLOCKSIZE           16
XCP_WK_BYTES                 32
XCP_ADMCTR_BYTES...........  16
XCP_ADM_QUERY        0x00010000
XCP_CPCOUNT                 128
XCP_CPID_BYTES                0  -- profiles are not currently supported
XCP_SPKISALT_BYTES.........   8
XCP_WKID_BYTES                8
XCP_FWID_BYTES               32
XCP_SERIALNR_CHARS            8
XCP_PIN_SALT_BYTES.........  16
```

```
XCP_CPBLOCK_BITS           128  -- block size of CP sets
XCP_CPBITS_MAX              65  -- largest supported CP bit (zero-based)
XCP_CSP_CONFIG_BYTES        40  -- CSP config structure bytecount (5.2.6.)
                                -- test-only feature, not supported
                                -- in regular production builds
XCP_DEV_MAX_DATABYTES     4096  -- upper limit on additional data bytes, for
                                -- SYS-TEST commands with aux. data
                                -- (arbitrary limit)
```

8.7.1. Query types

```
CK_IBM_XCPQ_API            0 -- API and build identifier (1.1.2)
CK_IBM_XCPQ_MODULE         1 -- module-level information (5.1.1)
CK_IBM_XCPQ_DOMAINS        2 -- list active domains & WKIDs (5.1.2)
CK_IBM_XCPQ_DOMAIN......... 3 -- domain information (5.1.3)
CK_IBM_XCPQ_SELFTEST       4 -- integrity & algorithm tests (5.1.4)
CK_IBM_XCPQ_EXT_CAPS       5 -- extended capabilities' count (8.7.1.1)
CK_IBM_XCPQ_EXT_CAPLIST    6 -- extended capabilities' listing (8.7.1.1)
reserved................... 7 -- reserved for future use
CK_IBM_XCPQ_AUDITLOG       8 -- audit record or records (5.4)
CK_IBM_XCPQ_DESCRTEXT      9 -- human-readable text/tokens
                             -- (5.1.1.1)
CK_IBM_XCPQ_EC_CURVES     10 -- supported elliptic curves,
                             -- bitmask (8.7.1.4)
CK_IBM_XCPQ_EC_CURVEGRPS  12 -- supported elliptic curves,
                             -- groups/categories, bitmask
                             -- (8.7.1.5)
CK_IBM_XCPQ_CP_BLACKLIST...13.-- control point blacklist:
                             -- control points which may
                             -- never be enabled due to
                             -- policy-minimum restrictions.
```

8.7.1.1. Query types: extended capabilities

Extended capability details are returned as a packed array of 32-bit raw integers, containing pairs of index-value pairs, similar to reporting administrative attributes (4.5.1) (the returned response is therefore always an even multiple of 32 bits).

See CK_IBM_XCPQ_EXT_CAPS and CK_IBM_XCPQ_EXT_CAPLIST queries for count and listing. The number of supported extended capabilities is also reported under module info queries (8.7.1)

```
reserved                      0  -- reserved for future use
reserved                      1  -- reserved for future use
CK_IBM_XCPXQ_AUDIT_EV_BYTES   2  -- largest audit event, bytecount
CK_IBM_XCPXQ_AUDIT_ENTRIES... 3..-- max. number of elements in event history
CK_IBM_XCPXQ_DEBUGLVL_MAX     4  -- backend diagnostics granularity
                                 -- 0 if backend is non-diagnostics build
CK_IBM_XCPXQ_ERRINJECT_FREQ   5  -- error-inject frequency N: N calls fail
                                 -- in a million with artificial errors
                                 -- 0 for production releases, which do not
                                 -- include error-injection (development only)
CK_IBM_XCPXQ_MULTIDATA_N      6  -- maximum number of supported
                                 -- sub-fields in multi-data
                                 -- requests. 0 if not supported,
                                 -- all-1's if no predefined limit
CK_IBM_XCPXQ_DOMIMPORT_VER... 7..-- 1-based revision of domain-import
                                 -- capability. 0 if feature
                                 -- is not supported
```

```
CK_IBM_XCPXQ_CERT_MAXBYTES    8..-- bytecount of largest accepted
                                 -- administrative certificate, if
                                 -- there is an upper limit.  0 if
                                 -- the backend does not enforce
                                 -- any specific limit of its own.
CK_IBM_XCPXQ_DOMIMPORT_VER... 7..-- number of module-internal dev
                                 -- counters supported, 0 if none.
                                 -- note that counter definitions are
                                 -- restricted to development use
reserved                     10  -- reserved for future use
reserved                     11  -- reserved for future use
CK_IBM_XCPXQ_MAX_SESSIONS    12
CK_IBM_XCPXQ_AVAIL_SESSIONS  13  -- maximum, currently available
                                 -- number of backend sessions
```

8.7.1.2. Query types: reserved host types

Types reserved for host use are at 0xff000000 or higher
(CK_IBM_XCP_HOSTQ_IDX).

```
CK_IBM_XCPHQ_COUNT        0xff000000 -- number of host-query indexes
                                     -- including this type itself
CK_IBM_XCPHQ_VERSION      0xff000001 -- host-specific package version, such as
                                     -- host library (package) version
CK_IBM_XCPHQ_VERSION_HASH 0xff000002 -- assumed-unique identifier of host code,
                                     -- such as version-identifying
                                     -- cryptographic hash (library signature
                                     -- field, version-control commit ID etc.)
CK_IBM_XCPHQ_DIAGS........0xff000003 -- host code diagnostic level
                                     -- 0 if host code is non-diagnostics build
CK_IBM_XCPHQ_HVERSION     0xff000004 -- human-readable host version ID
                                     -- recommended: encoded as UTF-8 string
CK_IBM_XCPHQ_TGT_MODE     0xff000005 -- host targeting modes
                                     -- returns supported target modes
                                     -- as bitmask
                                     -- if not available only compatibility
                                     -- target mode is in use
                                     -- See CK_IBM_XCPHQ_TGT_MODES_t
CK_IBM_XCPHQ_ECDH_DERPRM  0xff000006 -- support for ECDH1_DERIVE parameter
                                     -- returns non-zero value if available
                                     -- to be used in key derivation call to
                                     -- DeriveKey
```

8.7.1.3. Target support bitmask: Supported target modes of host library

Target modes are of type (CK_IBM_XCPHQ_TGT_MODES_t) and are returned
by a query of type (CK_IBM_XCPHQ_TGT_MODES_t). Systems not supporting
this query use a platform dependent target system. Please refer to your
system reference for information about the target concept.

```
CK_IBM_XCPHQ_TGT_MODES_TGTGRP   0x00000001
                                     -- target groups are supported
```

8.7.1.4. Query type: elliptic curve support

Since there is no standard PKCS11 query to enumerate supported elliptic
curves, we provide a bitmask enumerating curve IDs which the module
may support. Note that runtime-configured, effective set of curves
MAY differ, such as features controlled by control points.

```
The reported bitmask stores bits corresponding to EC curve identifiers
(8.1.1.5.) as a ‘‘big-endian bitmask’’ (6.18.) The first bit corresponds
to XCP_EC_C_NIST_P192 1.

8.7.1.5.  Query type: elliptic curve group support

Since there is no standard PKCS11 query to enumerate supported elliptic
curves, we provide a bitmask enumerating groups (categories) of EC
curves supported. Note that runtime-configured, effective set of curves
MAY differ, such as features controlled by control points.

The reported bitmask stores bits corresponding to EC curve identifiers
(8.1.1.5.) as a ‘‘big-endian bitmask’’ (6.18.) The first bit corresponds
to XCP_EC_CG_NIST 1.

Some of the indicated bits MAY themselves be polymorphic. As an example,
curves where ECDH and digital signatures are defined over different
representations (such as with Edwards curves) are reported as a single
curve group.

8.7.2.  Return values (CKR)

We provide the following vendor-defined PKCS#11 return values:

CKR_IBM_WKID_MISMATCH         0x80010001
CKR_IBM_INTERNAL_ERROR        0x80010002
CKR_IBM_TRANSPORT_ERROR       0x80010003
CKR_IBM_BLOB_ERROR............0x80010004
CKR_IBM_BLOBKEY_CONFLICT      0x80010005  -- WK setup changed during execution
                                          -- of a single request, preventing
                                          -- returnof encrypted data to the
                                          -- host. (i.e., an administrative
                                          -- operation changed WKs before the
                                          -- function could re-wrap data to pass
                                          -- back to the host
CKR_IBM_MODE_CONFLICT         0x80010006
CKR_IBM_NONCRT_KEY_SIZE       0x80010008  -- an RSA key in non-CRT form is
                                          -- encountered which is not supported
                                          -- by this hardware/engine
                                          -- configuration, if the setup has
                                          -- non/CRT-specific size restrictions.
                                          -- in essence, a more specific
                                          -- sub-division of the standard
                                          -- CKR_KEY_SIZE_RANGE, and MAY be
                                          -- safely mapped to that by host
                                          -- libraries.
                                          -- note that standard PKCS11
                                          -- GetMechanismInfo does not offer
                                          -- a way to report these differences.
CKR_IBM_WK_NOT_INITIALIZED....0x80010009
CKR_IBM_OA_API_ERROR          0x8001000a  -- unexpected/consistency error of
                                          -- CA (Outbound Auth) operations of
                                          -- the hosting HSM, if not otherwise
                                          -- classified.
CKR_IBM_REQ_TIMEOUT           0x8001000b  -- potentially long-running request,
                                          -- such as those involving prime
                                          -- generation, did not complete in a
                                          -- ‘‘reasonable’’ number of
                                          -- iterations. supported to prevent
```

```
                                                -- timeout-triggered module resets,
                                                -- such as mainframe firmware
                                                -- resetting modules ‘‘stuck in
                                                -- infinite loops’’ (as perceived by
                                                -- the host)
CKR_IBM_READONLY              0x8001000c  -- rejected due to backend
                                                -- persistent data in read-only state
CKR_IBM_STATIC_POLICY.........0x8001000d  -- request violates policy, will
                                                -- be rejected under all conditions
                                                -- by this backend. this is the
                                                -- the permanent form of policy
                                                -- failure (which is mapped to
                                                -- standard CKR_FUNCTION_CANCELED)
reserved                      0x8001000e  -- reserved for future use
reserved                      0x8001000f  -- reserved for future use
CKR_IBM_TRANSPORT_LIMIT       0x80010010  -- request is oversized due to
                                                -- transport (architecture)
                                                -- limitations
CKR_IBM_FCV_NOT_SET...........0x80010011  -- FCV of card not set
CKR_IBM_PERF_CATEGORY_INVALID 0x80010012 -- wrong, missing request performance
                                                -- category
CKR_IBM_API_MISMATCH          0x80010013 -- API ORDINAL number is unknown or
                                                -- function id is in illegal range
                                                -- host only return value
CKR_IBM_TARGET_INVALID        0x80010030 -- target token is invalid
                                                -- host only return value

8.7.3.  Custom mechanisms (CKM) and related constants

We reserve the following mechanisms for functions PKCS#11 does not
currently support (as of this writing):

CKM_IBM_CMAC              0x80010007
CKM_IBM_ECDSA_SHA224      0x80010008
CKM_IBM_ECDSA_SHA256      0x80010009
CKM_IBM_ECDSA_SHA384......0x8001000a
CKM_IBM_ECDSA_SHA512      0x8001000b
CKM_IBM_EAC               0x8001000d  -- extended access control (EAC)
                                      -- key derivation functions (6.8.1.3)
CKM_IBM_TESTCODE          0x8001000e
CKM_IBM_TRANSPORTKEY......0x80020005
CKM_IBM_SHA512_256        0x80010012  -- SHA-512/256, FIPS 180-4
CKM_IBM_SHA512_256_HMAC   0x80010014  -- HMAC with SHA-512/256
CKM_IBM_SHA512_256_KEY_DERIVATION
                          0x80010019
CKM_IBM_SHA512_224........0x80010013..-- SHA-512/224, FIPS 180-4
CKM_IBM_SHA512_224_HMAC   0x80010015  -- HMAC with SHA-512/224
CKM_IBM_SHA512_224_KEY_DERIVATION
                          0x8001001a

                                      -- hashed EC/DSA mechanisms:
                                      -- these vendor additions predate, but are
                                      -- functionally equivalent to the
                                      -- PKCS11 v2.40 proposed additions,
                                      -- CKM_ECDSA_SHA224 etc.
                                      --
CKM_IBM_ECDSA_SHA224......0x80010008  -- ECDSA, with SHA-224
CKM_IBM_ECDSA_SHA256      0x80010009  -- ECDSA, with SHA-256
CKM_IBM_ECDSA_SHA384      0x8001000a  -- ECDSA, with SHA-384
CKM_IBM_ECDSA_SHA512      0x8001000b  -- ECDSA, with SHA-512
```

```
SHA-3 variants (tentatively reserved, awaiting final PKCS v2.40+ formats):
                                      -- fixed-size outputs
CKM_IBM_SHA3_224          0x80010001  -- SHA-3, 224-bit digest
CKM_IBM_SHA3_224_HMAC     0x80010025  -- HMAC with SHA-3/224
CKM_IBM_SHA3_256          0x80010002  -- SHA-3, 256-bit digest
CKM_IBM_SHA3_256_HMAC     0x80010026  -- HMAC with SHA-3/256
CKM_IBM_SHA3_384..........0x80010003  -- SHA-3, 384-bit digest
CKM_IBM_SHA3_384_HMAC     0x80010027  -- HMAC with SHA-3/384
CKM_IBM_SHA3_512          0x80010004  -- SHA-3, 512-bit digest
CKM_IBM_SHA3_512_HMAC     0x80010028  -- HMAC with SHA-3/512
                                      -- extendable-output functions (XOFs)
                                      -- variable-sized, selectable output
```

```
SHA-3 variants (tentatively reserved, awaiting final PKCS v2.40+ formats):
                                      -- fixed-size outputs
CKM_IBM_SHA3_224          0x80010001  -- SHA-3, 224-bit digest
CKM_IBM_SHA3_224_HMAC     0x80010025  -- HMAC with SHA-3/224
CKM_IBM_SHA3_256          0x80010002  -- SHA-3, 256-bit digest
CKM_IBM_SHA3_256_HMAC     0x80010026  -- HMAC with SHA-3/256
CKM_IBM_SHA3_384..........0x80010003  -- SHA-3, 384-bit digest
CKM_IBM_SHA3_384_HMAC     0x80010027  -- HMAC with SHA-3/384
CKM_IBM_SHA3_512          0x80010004  -- SHA-3, 512-bit digest
CKM_IBM_SHA3_512_HMAC     0x80010028  -- HMAC with SHA-3/512
                                      -- extendable-output functions (XOFs)
                                      -- variable-sized, selectable output
```

```
CKM_IBM_EC_X25519         0x8001001b..-- curve25519 EC/DH
CKM_IBM_EC_X25519_RAW     0x80010029..-- curve25519 EC/DH, with KEK
CKM_IBM_ED25519_SHA512    0x8001001c  -- EdDSA/SHA-512, with ed25519,
                                      -- without pre-hashing


CKM_IBM_EC_X448           0x8001001e..-- curve448 (Goldilocks), key agreement
CKM_IBM_EC_X448_RAW       0x80010030..-- curve448 (Goldilocks), key agreement,
                                      -- with KEK
CKM_IBM_ED448_SHA3........0x8001001f  -- ed448 signatures, with SHA-3/XOF
                                      -- without prehashing
CKM_IBM_CPACF_WRAP        0x80060001  -- import a blob as an protected key
                                      -- blob needs XCP_BLOB_PROTKEY_EXTRACTABLE


CKM_IBM_DILITHIUM         0x80010023  -- PQ algorithms: Dilithium
The following key type is introduced with the Dilithium mechanism:
CKK_IBM_PQC_DILITHIUM     0x80010023 -- PQ algorithms: Dilithium key type

We will revisit some of the currently vendor-defined mechanisms if
they get standardized in the future. EdDSA and SHA3
mechanisms are possible candidates as of this writing. Future, standard
identifiers will continue to coexist with our vendor additions.

8.7.4.  Attribute types reported by keytype/mechanism-attribute query

XCP_CKA_B_NEED           1  -- Boolean, must be present
XCP_CKA_B_OPT            2  -- Boolean, optionally present
XCP_CKA_B_NOT            3  -- Boolean, must NOT BE specified
                            -- not expected to appear in queries
XCP_CKA_B_OPT_DEFTRUE    4  -- Boolean, optional, default TRUE
XCP_CKA_B_OPT_DEFFALSE.. 5..-- Boolean, optional, default FALSE
```

```
XCP_CKA_B_QUERY          6  -- Boolean, query only

XCP_CKA_U32_NEED........ 7..-- Integer/32, must be present
XCP_CKA_U32_OPT          8  -- Integer/32, optionally present
XCP_CKA_U32_NOT          9  -- Integer/32, MUST NOT be specified
                            -- not expected to appear in queries
XCP_CKA_U32_QUERY       10  -- Integer/32, query only

XCP_CKA_VARLEN_NEED.....11..-- variable-length, must be present
XCP_CKA_VARLEN_OPT      12  -- variable-length, optional
XCP_CKA_VARLEN_NOT      13  -- variable-length, MUST NOT be specified
                            -- not expected to appear in queries
XCP_CKA_VARLEN_QUERY    14  -- variable-length, query only

Attributes marked as ‘‘query only’’ or ...NOT are included for
documentation purposes; they are not expected to be specified during key
generation or import. Query-only attributes may become available when a
key is created.

Types marked as ...NOT are sensitive or intentionally not accessible,
and are explicitly reported to mark them as unavailable. Host code MUST
NOT pass them to backend during key generation, and MAY refuse to even
query them. Standard attributes involving secret or private-key fields
are relevant examples for such classifications.

8.8.  Referenced PKCS11 constants

We reference certain PKCS#11 v2.20/v2.40 constants in this document,
replicating relevant ones here for consistency:

CKA_KEY_TYPE                   0x00000100
CKA_VALUE_LEN                  0x00000161
CKA_CHECK_VALUE                0x00000090
CKM_DSA_PARAMETER_GEN..........0x00002000
CKM_AES_CBC                    0x00001082
CKM_AES_CBC_PAD                0x00001085
CKM_DES3_CBC                   0x00000133
CKM_DES3_CBC_PAD...............0x00000136
CKM_DES_CBC                    0x00000122
CKM_DES_CBC_PAD                0x00000125

                               -- SHA-224-related constants, not in PKCS11
                               -- v2.20, added in v2.20 amendment 3,
                               -- finalized in official v2.40
CKM_SHA224                     0x00000255
CKG_MGF1_SHA224                0x00000005
CKM_SHA224_HMAC                0x00000256
CKM_SHA224_HMAC_GENERAL        0x00000257
CKM_SHA224_RSA_PKCS            0x00000046
CKM_SHA224_RSA_PKCS_PSS        0x00000047
CKM_SHA224_KEY_DERIVATION      0x00000396

                               -- SHA-512/nnn variants: FIPS 186-4 truncated
                               -- SHA-512 hashes, added in PKCS11 v2.40
CKM_SHA512_224                 0x00000048
CKM_SHA512_224_HMAC            0x00000049
CKM_SHA512_224_HMAC_GENERAL    0x0000004a
CKM_SHA512_224_KEY_DERIVATION..0x0000004b
CKM_SHA512_256                 0x0000004c
CKM_SHA512_256_HMAC            0x0000004d
```

```
CKM_SHA512_256_HMAC_GENERAL    0x0000004e
CKM_SHA512_256_KEY_DERIVATION  0x0000004f

Some of these mechanism constants have vendor-extended custom equivalents,
which predate their v2.40-specified equivalents (see 8.7.3).

8.9.  Audit-related constants

8.9.1.  Audit event types

XCP_LOGEV_QUERY           0x00000000
XCP_LOGEV_FUNCTION        0x00000001
XCP_LOGEV_ADMFUNCTION     0x00000002
XCP_LOGEV_STARTUP.........0x00000003
XCP_LOGEV_SHUTDOWN        0x00000004
XCP_LOGEV_SELFTEST        0x00000005
XCP_LOGEV_DOM_IMPORT      0x00000006
XCP_LOGEV_DOM_EXPORT......0x00000007
XCP_LOGEV_FAILURE         0x00000008
XCP_LOGEV_GENERATE        0x00000009
XCP_LOGEV_REMOVE          0x0000000a
XCP_LOGEV_SPECIFIC........0x0000000b
                                    -- specific log event
                                    -- see detailed list (8.9.2)
XCP_LOGEV_STATE_IMPORT    0x0000000c
XCP_LOGEV_STATE_EXPORT    0x0000000d

8.9.2.  Specific audit events

These values may show up under ‘‘event details’’ of audit records.

XCP_LOGSPEV_TRANSACT_ZEROIZE  0xffff0001 -- pending transaction forced module
                                         -- to zeroize (such as: import failed
                                         -- in inconsistent intermediate state)
XCP_LOGSPEV_KAT_FAILED        0xffff0002 -- algorithm known-answer tests failed
XCP_LOGSPEV_KAT_COMPLETED     0xffff0003 -- algorithm known-answer tests passed
XCP_LOGSPEV_EARLY_Q_START.....0xffff0004 -- start of early-audit events:
                                         -- subsequent events have proper
                                         -- order, show only approximate time
XCP_LOGSPEV_EARLY_Q_END       0xffff0005 -- end of early-audit events:
                                         -- subsequent events show exact time
XCP_LOGSPEV_AUDIT_NEWCHAIN    0xffff0006 -- audit chain was corrupted; removed,
                                         -- generating new instance,
                                         -- starting new chain
XCP_LOGSPEV_TIMECHG_BEFORE    0xffff0007 -- time change: original time
XCP_LOGSPEV_TIMECHG_AFTER.....0xffff0008 -- time change: updated time
XCP_LOGSPEV_MODSTIMPORT_START 0xffff0009 -- accepted full-state import
                                         -- data structure, starting update
XCP_LOGSPEV_MODSTIMPORT_FAIL  0xffff000a -- rejected import structure
                                         -- issued after initial verify
                                         -- indicates some inconsistency
                                         -- of import data structures
XCP_LOGSPEV_MODSTIMPORT_END   0xffff000b -- completed full-state import

8.9.3.  Audit event flags

XCP_LOGFL_WK_PRESENT        0x80000000
XCP_LOGFL_FINALWK_PRESENT   0x20000000
XCP_LOGFL_KEYREC0_PRESENT   0x10000000
XCP_LOGFL_KEYREC1_PRESENT...0x04000000
```

```
XCP_LOGFL_KEYREC2_PRESENT   0x02000000
XCP_LOGFL_FINTIME_PRESENT   0x01000000
XCP_LOGFL_SALT0_PRESENT     0x00800000
XCP_LOGFL_SALT1_PRESENT.....0x00400000
XCP_LOGFL_SALT2_PRESENT     0x00200000

8.10.  Function sub-variants

Not all variants are supported by all functions.

XCP_FNVAR_SIZEQUERY         1     -- sizequery: data bytecount[BE64] ->
                                  --            response bytecount[64]
XCP_FNVAR_MULTIDATA         2     -- multi-data request (see 14.1)
XCP_FNVAR_MULTISIZEQ        3     -- multi-data request, size query (1.4.4.)


9.  Command parameter lists

Parameter layout is defined within Requests (1) and Responses (2).
The following listing enumerates input parameter(s), and response field(s).
System parameters (1) are excluded.

Parameters labeled with ``(INT)'' are integers with size guarantees
(up to 32 bits).

For bytecounts replacing actual data for size queries, see (1.2).
Parameter annotations show when a field changes meaning with
a size query (``sq:...'').

9.1.  Key generation

  GenerateKey
    inputs   [5]
      1.  variant               (1: size query)  (INT)
      2.  key bytes             (unused if not needed)
      3.  key mech              (CKM_... format, packed)
      4.  attributes            (internal READ representation: mask == bits)
      5.  pin blob              (optional)
    outputs   [2]
      1.  key blob              (sq: blob bytecount)
      2.  PKCS11 checksum       (at least 24 bits)

  GenerateKeyPair
    inputs   [4]
      1.  key algorithm          (CKM... format, integer)  (INT)
      2.  attributes, public     (internal representation)
      3.  attributes, private    (internal representation)
      4.  pin blob               (empty if unused)
    outputs   [2]
      1.  key blob               (private key object)
      2.  SPKI                   (publickeyinfo, MACed)

  DeriveKey
    inputs   [5]
      1.  derivation mechanism (packed)
      2.  new key attributes
      3.  base key (blob) (raw non-blob with certain formats)
      4.  pin blob              (optional; for session-bound objects only)
      5.  auxiliary data (or key blob, depending on mechanism)
    outputs   [2]
```

```
        1.  derived key (blob)
        2.  PKCS11 checksum || derived bitcount[BE32]

  9.2.  Digesting

    DigestInit
      inputs   [2]
        1.  variant                   (1: size query)  (INT)
        2.  digest mechanism
      output
        1.  initialized digest state  (sq: state bytecount)

    Digest
      inputs   [3]
        1.  variant         (1: size query)  (INT)
        2.  digest state    (contains mechanism inside)
        3.  data            (sq: input bytecount)
      output
        1.  digest          (sq: digest bytecount)

    DigestKey
      inputs   [2]
        1.  digest state        (contains mechanism)
        2.  key blob (object)
      output
        1.  updated state

    DigestUpdate
      inputs   [2]
        1.  digest state     (contains mechanism)
        2.  data
      output
        1.  updated state

    DigestFinal
      inputs   [2]
        1.  variant          (1: size query)  (INT)
        2.  digest state
      output
        1.  digest           (sq: digest bytecount)

    DigestSingle
      inputs   [3]
        1.  variant            (1: size query)  (INT)
        2.  digest mechanism
        3.  data               (sq: input bytecount)
      output
        1.  digest             (sq: digest bytecount)

  9.3.  Signing and verification

    SignInit
      inputs   [3]
        1.  variant                 (1: size query)  (INT)
        2.  sign mechanism
        3.  key blob
      output
        1.  initialized sign state  (sq: state bytecount)

    Sign
  inputs   [3]
    1.  variant            (1: size query)  (INT)
    2.  sign state
    3.  data
  output
    1.  signature or MAC   (sq: signature/MAC bytecount)

SignUpdate
  inputs   [2]
    1.  sign state
    2.  data
  output
    1.  updated state

SignFinal
  inputs   [2]
    1.  variant        (1: size query)  (INT)
    2.  sign state
  output
    1.  signature      (sq: signature bytecount)

SignSingle
  inputs   [4]
    1.  variant           (1: size query)  (INT)
    2.  sign mechanism
    3.  key blob (object)
    4.  data              (sq: data bytecount)
  output
    1.  signature         (sq: signature bytecount)

VerifyInit
  inputs   [3]
    1.  variant                           (1: size query)  (INT)
    2.  verify mechanism
    3.  key blob (symmetric) or raw/MACed SPKI (PK verify)
  output
    1.  initialized verify state          (sq: state bytecount)

Verify
  inputs   [3]
    1.  verify state
    2.  signature
    3.  data
  (no data output)

VerifyUpdate
  inputs   [2]
    1.  verify state
    2.  data
  output
    1.  updated state

VerifyFinal
  inputs   [2]
    1.  verify state
    2.  signature
  (no data output)

VerifySingle
  inputs   [4]
```

```
        1.  mechanism
        2.  public key (SPKI) or blob (symmetric mech)
        3.  data
        4.  signature
      (no data output)

9.4.  Encryption and decryption

    DecryptInit
      inputs   [3]
        1.  variant                     (1: size query)  (INT)
        2.  mechanism
        3.  key blob
      output
        1.  initialized decrypt state   (sq: output bytecount)

    Decrypt
      inputs   [3]
        1.  variant            (1: size query)  (INT)
        2.  decrypt state blob
        3.  ciphertext         (sq: ciphertext bytecount)
      output
        1.  plaintext          (sq: plaintext bytecount)

    DecryptUpdate
      inputs   [3]
        1.  variant               (1: size query)  (INT)
        2.  decrypt state
        3.  ciphertext, increment (not a bytecount even for size query)
      outputs   [2]
        1.  updated state         (empty if unchanged)
        2.  plaintext, increment  (empty if none produced) (sq: written size)

    DecryptFinal
      inputs   [2]
        1.  variant           (1: size query)  (INT)
        2.  state blob
      output
        1.  decrypted output  (sq: output bytecount) (empty if none produced)

    DecryptSingle
      inputs   [4]
        1.  variant       (1: size query)  (INT)
        2.  mechanism
        3.  key blob
        4.  ciphertext    (sq: ciphertext bytecount)
      output
        1.  plaintext     (sq: plaintext bytecount)

    EncryptInit
      inputs   [3]
        1.  variant                     (1: size query)  (INT)
        2.  mechanism
        3.  key blob
      output
        1.  initialized encrypt state   (sq: output bytecount)

    Encrypt
      inputs   [3]
        1.  variant              (1: size query)  (INT)
        2.  encrypt state blob
        3.  plaintext            (sq: plaintext bytecount)
      output
        1.  ciphertext           (sq: ciphertext bytecount)

    EncryptUpdate
      inputs   [3]
        1.  variant               (1: size query)  (INT)
        2.  encrypt state
        3.  plaintext, increment  (not a bytecount even for size query)
      outputs   [2]
        1.  updated state         (empty if unchanged)
        2.  ciphertext, increment (empty if none produced) (sq: written size)

    EncryptFinal
      inputs   [2]
        1.  variant           (1: size query)  (INT)
        2.  state blob
      output
        1.  encrypted output  (sq: output bytecount) (empty if none produced)

    EncryptSingle
      inputs   [4]
        1.  variant       (1: size query)  (INT)
        2.  mechanism
        3.  key blob
        4.  plaintext     (sq: plaintext bytecount)
      output
        1.  ciphertext    (sq: ciphertext bytecount)

  Note that EncryptSingle also implements development-only functions (5.2),
  if they are supported.

  9.5.  Random numbers

    SeedRandom
      input
        1.  seed to mix, must not be empty
      (no data output)

    GenerateRandom
      input
        1.  number of random bytes requested  (32-bit raw integer)  (INT)
      output
        1.  random bytes

  9.6.  Key transport

    UnwrapKey
      inputs   [6]
        1.  new key's attributes (ignored for attribute-bound keys)
        2.  unwrapping mechanism
        3.  wrapping KEK blob    (ignored when adding MAC to SPKIs)
        4.  MAC key              (optional; attribute-bound mech/s only)
        5.  pinblob              (optional)
        6.  wrapped data         (contains SPKI when adding MAC)
      outputs   [2]
        1.  unwrapped key (blob)
        2.  checksum (symm keys) or SPKI (private keys), followed by bitcount
    WrapKey
      inputs   [5]
        1.  variant          (1: size query)  (INT)
        2.  mechanism
        3.  blob to wrap
        4.  wrapping blob (KEK)
        5.  MAC key          (attribute-bound mech only; empty otherwise)
      output
        1.  wrapped key      (sq: wrapped object bytecount)
```

9.7. Queries

```
  GetAttributeValue
    inputs   [2]
      1.  key (blob)
      2.  attribute list
    output
      1.  packed attribute values

  GetMechanismInfo
    input
      1.  mechanism
    output
      1.  three big-endian 32-bit integers: minKeySize, maxKeySize, flags
```

Note that EP11 reports minKeySize and maxKeySize for symmetric and HMAC mechanisms in bytes and for asymmetric mechanisms in bits.

```
  GetMechanismList
    (no input)
    output
      1.  sequence of 32-bit, big-endian mechanisms

  get_xcp_info
    inputs   [2]
      1.  query type  (INT)
      2.  subquery
    output
      1.  packed info structure
```

9.8. Administration

```
  SetAttributeValue
    inputs   [2]
      1.  blob (key object)
      2.  attribute field
    output
      1.  updated blob

  admin
    inputs   [2]
      1.  payload: full administrative command block
      2.  signature/s, if present
    outputs   [2]
      1.  response payload
      2.  signature, if present
```

9.8.1. Session management

```
  Login
```

```
    inputs   [2]
      1.  PIN/passphrase
      2.  nonce, perturbs PIN for session ID generation  (optional)
    output
      1.  PIN blob (XCP_WK_BYTES at start is session ID)

  Logout
    input
      1.  PIN blob
    (no data output)

9.9.  Other calls

  ReencryptSingle
    inputs   [6]
      1.  variant                  (INT)
      2.  decrypt (1st) mechanism
      3.  encrypt (2nd) mechanism
      4.  decrypting (1st) key blob
      5.  encrypting (2nd) key blob
      6.  data or data len (if querying size)
    output
      1.  de+encrypted output

10.  Version dependent changes

10.1.  GetMechanismInfo - HMAC min/maxKeySize

HMAC mechanisms report minKeySize in bytes and maxKeySize in bits for following
module firmware versions, up to and including.
CEX6S: 3.6.8
       3.5.11

CEX5S: 2.6.2
       2.5.5
       2.4.19

More recent module firmware versions report both minKeySize and maxKeySize for
HMAC mechanisms in bytes.

10.2. API ordinal

10.2.1. API ordinal 3 - DeriveKey

With an API ordinal 3, modules (only CEX7S) introduced CK_ECDH1_DERIVE_PARAMS
for CKM_ECDH1_DERIVE as mechanism parameter.  The previous variant of passing
the plain public key as parameter remains available under API ordinal 2. This
change is included starting from firmware version:

CEX7S: 4.7.9

10.2.2. API ordinal 4 - Protected key import

Support for protected key import is limited to modules with an API ordinal of 4
or higher. The corresponding change is also indicated by the presence of the
CK_IBM_DOM_PROTKEY_ALLOW domain flag (6.13.2).
This change is included in starting from firmware version:

CEX7S: 4.7.14
```

```
Modules without API ordinal 4 may still tolerate related blob
attributes. Such support is indicated by presence of
CKF_IBM_HW_PROTKEY_TOLERATION in the extended flags (6.13.1). This support is
available starting from firmware versions:

CEX7S: 4.7.14
CEX6S: 3.7.8
CEX5S: 2.7.8
```

Other references

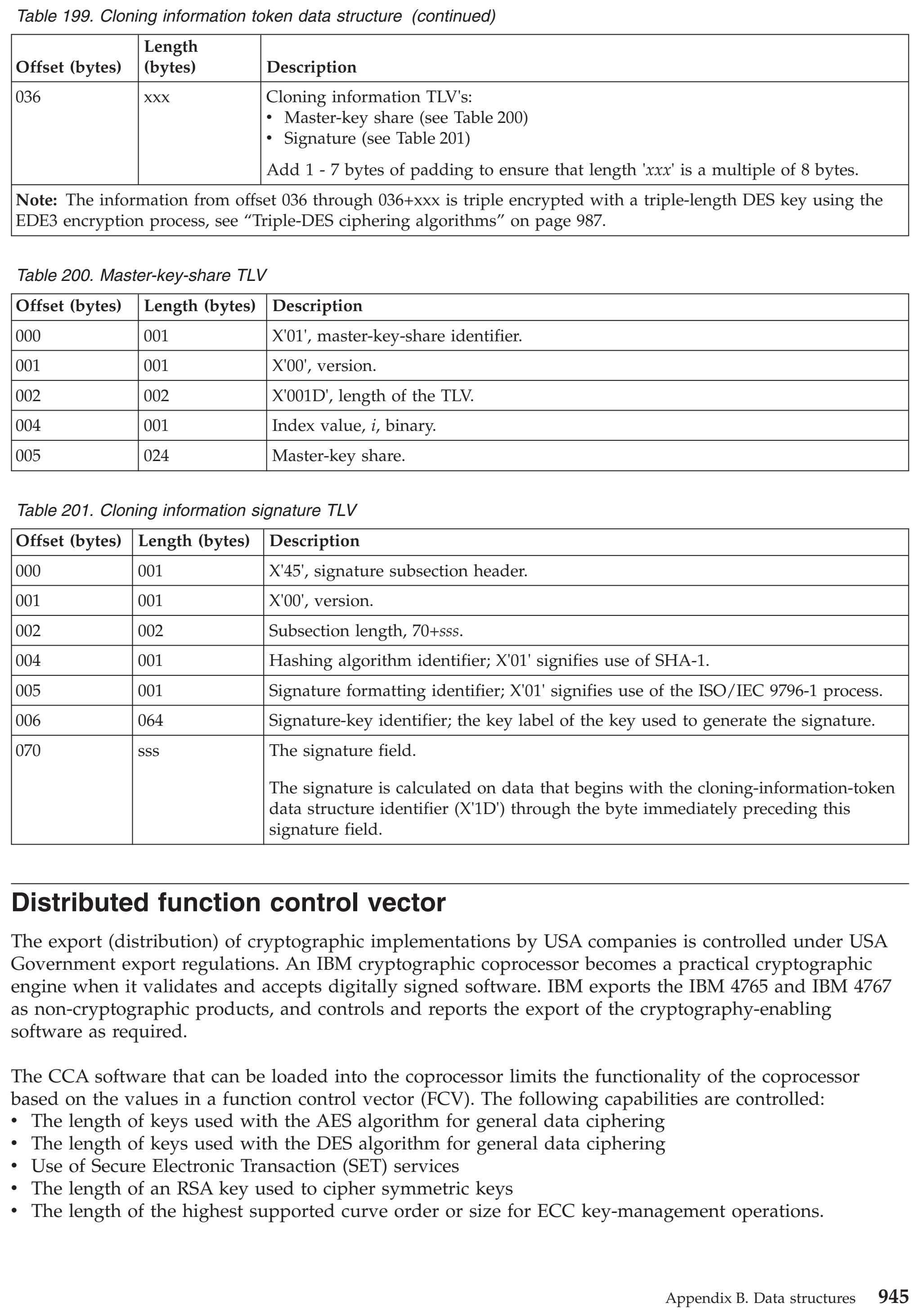

Table 199. Cloning information token data structure (continued)

Offset (bytes)	Length (bytes)	Description
036	xxx	Cloning information TLV's: • Master-key share (see Table 200) • Signature (see Table 201) Add 1 - 7 bytes of padding to ensure that length '*xxx*' is a multiple of 8 bytes.
Note: The information from offset 036 through 036+xxx is triple encrypted with a triple-length DES key using the EDE3 encryption process, see "Triple-DES ciphering algorithms" on page 987.		

Table 200. Master-key-share TLV

Offset (bytes)	Length (bytes)	Description
000	001	X'01', master-key-share identifier.
001	001	X'00', version.
002	002	X'001D', length of the TLV.
004	001	Index value, *i*, binary.
005	024	Master-key share.

Table 201. Cloning information signature TLV

Offset (bytes)	Length (bytes)	Description
000	001	X'45', signature subsection header.
001	001	X'00', version.
002	002	Subsection length, 70+*sss*.
004	001	Hashing algorithm identifier; X'01' signifies use of SHA-1.
005	001	Signature formatting identifier; X'01' signifies use of the ISO/IEC 9796-1 process.
006	064	Signature-key identifier; the key label of the key used to generate the signature.
070	sss	The signature field. The signature is calculated on data that begins with the cloning-information-token data structure identifier (X'1D') through the byte immediately preceding this signature field.

Distributed function control vector

The export (distribution) of cryptographic implementations by USA companies is controlled under USA Government export regulations. An IBM cryptographic coprocessor becomes a practical cryptographic engine when it validates and accepts digitally signed software. IBM exports the IBM 4765 and IBM 4767 as non-cryptographic products, and controls and reports the export of the cryptography-enabling software as required.

The CCA software that can be loaded into the coprocessor limits the functionality of the coprocessor based on the values in a function control vector (FCV). The following capabilities are controlled:

- The length of keys used with the AES algorithm for general data ciphering
- The length of keys used with the DES algorithm for general data ciphering
- Use of Secure Electronic Transaction (SET) services
- The length of an RSA key used to cipher symmetric keys
- The length of the highest supported curve order or size for ECC key-management operations.

| IBM distributes the FCV in a digitally signed data structure (certificate). Table 202 shows the format of
| the data structure that contains the FCV as distributed by IBM. Table 202 shows that the FCV is located at
| offset 1,238 (X'4D6'), and has a length of 588 bytes for an IBM 4765 and a length of 208 for an IBM 4767.
| This information is needed for loading an FCV using the Cryptographic_Facility_Control verb.

For information on loading or clearing an FCV, refer to the "Cryptographic_Facility_Control (CSUACFC)" on page 52. For information on querying an FCV, refer to the "Cryptographic_Facility_Query (CSUACFQ)" on page 60.

Note:

1. Government policies and the FCV do not limit the key-length of keys used in digital signature operations.
2. The SET services can employ 56-bit DES for data encryption, and 1024-bit RSA key-lengths when distributing DES keys.

| *Table 202. FCV distribution structure, FCV format version X'01'*

Offset decimal (hex)	**Length decimal (in bytes)**	**Description**
000 (000)	1,158	Package header and validating-key public key certificate.
1,158 (486)	080	Descriptive text coded in ASCII.
1,238 (4D6)	588	Function Control Vector (FCV) An FCV structure (defined below) consists of a header followed by a concatenation of cryptographic enablement information and a digital signature signed by IBM. The signature along with other data contained in the FCV must pass validation checking within the coprocessor in order to be accepted and activated. To load the FCV, use the CSUACFC verb or, alternatively, use the Cryptographic Node Management (CNM) utility, described in the *IBM 4767 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual*. To use CSUACFC, call the verb with the **LOAD-FCV** rule-array keyword and use the *verb_data* parameter to identify the FCV structure, and the *verb_data_length* parameter to identify the length. The length depends on which IBM cryptographic coprocessor is being loaded: **Coprocessor** **FCV length in bytes, *fff*** **IBM 4765** 588 (X'024C') **IBM 4767** 208 (X'00D0')
FCV structure of IBM 4765 and IBM 4767 PCIe Cryptographic Coprocessors (offset 1238 above)		
FCV header		
000 (000)	001	Record ID (X'06').
001 (001)	001	Header Version (X'00').
002 (002)	002	Reserved X'0000'.

Table 202. FCV distribution structure, FCV format version X'01' (continued)

Offset decimal (hex)	**Length decimal (in bytes)**	**Description**
004 (004)	004	Total FCV structure length in bytes (little endian format) The total FCV structure length includes the digital signature. **Coprocessor** **Value** **IBM 4765** X'4C020000' (588) **IBM 4767** X'D0000000' (208)
008 (008)	004	Signature rules (little endian format)The signature rules identify how the signature was created: **Coprocessor** **Value** **IBM 4765** X'FF000000' (0255) / RSA signature **IBM 4767** X'000F0000' (3840) / ECC signature (3840) - IBM 4767 only
FCV cryptographic enablement information		
012 (00C)	001	FCV format version (X'01').
013 (00D)	001	CCA services class byte (currently ignored): **Value** **Meaning** **X'00'** Basic CCA services not enabled **X'01'** Basic CCA services enabled

Table 202. FCV distribution structure, FCV format version X'01' (continued)

Offset decimal (hex)	Length decimal (in bytes)	Description
014 (00E)	001	Symmetric algorithm enablement flag byte Symmetric algorithm enablement: **Value** **Meaning** **B'00xx xxxx'** Reserved AES encryption and decryption enablement: **Value** **Meaning** **B'xx00 0xxx'** AES encryption and decryption not enabled **B'xx00 1xxx'** 128-bit AES encryption and decryption enabled **B'xx01 0xxx'** Undefined **B'xx01 1xxx'** 192-bit and 128-bit AES encryption and decryption enabled **B'xx10 0xxx'** Undefined **B'xx10 1xxx'** Undefined **B'xx11 0xxx'** Undefined **B'xx11 1xxx'** 256-bit, 192-bit, and 128-bit AES encryption and decryption enabled Triple-DES encryption and decryption enablement: **Value** **Meaning** **B'xxxx x0xx'** Triple-DES encryption and decryption not enabled **B'xxxx x1xx'** Triple-DES encryption and decryption enabled 56-bit DES encryption and decryption enablement: **Value** **Meaning** **B'xxxx xx0x'** 56-bit DES encryption and decryption not enabled **B'xxxx xx1x'** 56-bit DES encryption and decryption enabled CDMF encryption and decryption enablement (formerly used on IBM 4758; currently ignored): **Value** **Meaning** **B'xxxx xxx0'** CDMF encryption and decryption not enabled **B'xxxx xxx1'** CDMF encryption and decryption enabled
015 (00F)	001	Secure Electronic Transaction (SET) services enablement byte: **Value** **Meaning** **X'00'** SET services not enabled **X'01'** SET services enabled (CSNBSBC and CSNBSBD 56-bit DES and 1024-bit RSA key lengths permitted)
016 (010)	004	Reserved (X'00000000').

Table 202. FCV distribution structure, FCV format version X'01' (continued)

Offset decimal (hex)	Length decimal (in bytes)	Description
020 (014)	002	Maximum supported modulus bit length for encryption and decryption of symmetric keys (little endian format): **Value** **X'0002' (512)** **X'0004' (1024)** **X'0008' (2048)** **X'0010' (4096)**
022 (016)	002	Maximum supported curve order or size in bits for elliptic curve cryptography (ECC) key-management operations (little endian format): **IBM 4765 Value** X'0902' (521) **IBM 4767 Value** X'A000' (160) X'C000' (192) X'E000' (224) X'0001' (256) X'4001' (320) X'8001' (384) X'0002' (512) X'0902' (521)
024 (018)	052	Reserved (X'00...00').
FCV digital signature		
076 (04C)	sss	FCV digital signature This value is calculated on the FCV structure, starting with the record ID of the FCV header up to but not including the digital signature itself. The digital signature is signed as follows: • For the IBM 4765, the digital signature is signed by a 4096-bit RSA key FcvPuK using the ANS X9.31 digital signature hash formatting method for a length of 512 bytes. • For the IBM 4767, the digital signature is signed by a 521-bit ECC key FcvPuK using ECDSA for a length of signature is 132 bytes.

Visa Format-Preserving Encryption supporting information

The Visa Format-Preserving Encryption (VFPE) Option has an algorithm that uses an alphabet parameter. An alphabet assigns a sequential number set for all potential characters for a given field type that is used in the conversion of payment card data prior to encryption. VFPE applies to these verbs:

- FPE_Decrypt (CSNBFPED)
- FPE_Encrypt (CSNBFPEE)
- FPE_Translate (CSNBFPET)
- Encrypted_PIN_Translate_Enhanced (CSNBPTRE)

These CCA verbs convert payment card data as required to or from VFPE alphabet numbers as determined by rule-array keyword. The alphabet tables below are meant to provide a reference for the valid set of characters for each of the four Visa payment card data formats (namely, PAN, Cardholder Name, Track 1 Discretionary Data, and Track 2 Discretionary Data).

VFPE payment card data can be in any one of these formats:

Glossary

AB also *Attribute-bound,* a proprietary extension to PKCS#11, implemented as a vendor Boolean attribute. Keys marked as AB may not be separated from their usage restrictions or other attributes, even if transported. AB key transport is always authenticated; all participating keys—key, key-encrypting key, and authentication key—must be attribute-bound.

Since the AB property is just an extended restriction of key transport, AB objects may be used interchangeably with non-AB objects by other functional services (those not related to un/wrapping).

BBRAM Battery-backed RAM

CDU Concurrent (Driver) Update, the capability of updating firmware while existing applications continue to run within IBM HSMs. Possible due to module-internal redundancy: separate processors run OS/applications and administrative code.

CP Control Point, access or usage-restriction settings represented as a single, per-domain bitvector

CSP Cryptographic Service Provider

domain module-internal key virtualization unit: an internal unit with its own set of administrators, policy settings, and keys. It is identified by a domain index.

DRNG Deterministic-Random Number Generator, deterministic postprocessing expanding entropy (TRNG) output into a pseudorandom stream

EP11 module An EP11 Module is a HSM configured for EP11 and is identified by a module number. It is also simply refered as module in this document.

FCV Function Control Vector, an IBM-issued infrastructure data structure, restricting available algorithms. FCVs are used, as an example, to enforce export-control restrictions where applicable.

HSM Hardware Security Module

IID Independent and Identically Distributed, the assumed property of raw-entropy sources feeding the conditioned TRNG used by backends

KAT Known Answer Test

OA *Outbound Authentication*, a public-key based authentication mechanism. Allows untampered cards to prove their integrity, present ownership of the *device keypair*, and trace it back to the IBM Factory CA. OA keys are not represented as PKCS#11 objects, and their validity is established offline, outside HSMs.

POST Power-On Self-Test, infrastructure tests resident in ROM and flash, executed during startup, before OS/application startup. Successful OS/application startup implies that base POST tests succeeded. A significant subset of POST self-tests, primarily algorithmic KATs, may be invoked with EP11 itself.

RAS Abbreviation of *Reliability, Availability, Serviceability,* features specifically added for resiliency and data assurance

RNG Random Number Generator, further specialized as true-random entropy source (TRNG) or deterministic-random (DRNG) post-processing

SKI `SubjectKeyIdentifier`, an "almost unique" identifier of a public key, generally, a hash of a key-unique parameter. We follow a standard solution, and calculate a hash of the BIT STRING `subjectPublicKey` of the SPKI [RHPFS02, 4.2.1.2]. We currently use SHA-256 as a hash function.

SPKI `SubjectPublicKeyInfo`, a collection of self-describing, industry standard binary formats for public keys. SPKI structures contain type information, unambiguously identifying key types and parameters. RSA SPKIs are described in [SKH05, 1.2], EC ones in [TBY$^+$09, 2.1].

PKCS#8 private key Industry-standard serialization format for private keys [Tur10]; RSA private keys are described in [JK03, A.1.2], EC ones in [TB10, 3]. We defined proprietary extensions for to non-Weierstrass curves; these are described in our wire specification.

Restrictions on the use of PKCS#8 structures specific to PKCS#11 are documented in [PKC15b, 2.5].

PKCS#11 key checksum A 24-bit checksum, corresponding to the PKCS#11 attribute `CKA_CHECK_VALUE`, allowing easy disambiguation between different keys (equality may not be uniquely determined, due to collisions in the short checksum). The checksum-construction algorithm depends on keytype.

We extend the checksum concept to public keys, reporting the most significant bits of the key SKI as checksum. As a side effect, corresponding public and private keys will be identifiable, with matching SKIs.

target group A set of zero or more targets maintained by the host library which can be used for load balancing and failsafe reasons. A zero target group implies all targets available to the system.

target A single module/domain combination that is represented by a module number and a domain index.

target token Target tokens identify a target or a group of targets. Target tokens are for example created when modules are registered by the host library and used as parameter in all EP11 host functions that interact with EP11 modules.

TRNG True-Random Number Generator, an entropy source

WK Wrapping Key, our terminology for domain-specific keys encrypting an externally stored keystore

References

[AAD+09] Jeff Arnold, Tim Abbott, Waseem Daher, Gregory Price, Nelson Elhage, Geoffrey Thomas, and Anders Kaseorg. Security impact ratings considered harmful. *ArXiv e-prints (Arxiv 0904.4058)*, April 2009.

[ADG+92] D. F. Ackerman, M. H. Decker, J. J. Gosselin, K. M. Lasko, M. P. Mullen, R. E. Rosa, E. V. Valera, and B. Wile. Simulation of IBM Enterprise System/9000 models 820 and 900. *IBM Journal of Research and Development*, 36(4):751–764, July 1992.

[AP15] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. Cryptology ePrint Archive, Report 2015/1129, `eprint.iacr.org/2015/1129`, November 2015. [accessed 2016-01-05].

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, 2016. USENIX Association.

[ASVH13] Bernhard Amann, Robin Sommer, Matthias Vallentin, and Seth Hall. No attack necessary: the surprising dynamics of SSL trust relationships. In Charles N. Payne Jr., editor, *Proceedings of the Annual Computer Security Applications Conference, ACSAC'13, New Orleans, LA, USA, December 9-13, 2013*, pages 179–188. ACM, 2013.

[AVHS12] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Extracting certificates from live traffic: A near realtime SSL notary service. Technical Report TR-12-014, International Computer Science Institute (ICSI), University of California, Berkeley, November 2012.

[BA01] Mike Bond and Ross J. Anderson. API-level attacks on embedded systems. *IEEE Computer*, 34(10):67–75, 2001.

[Bai04] David H. Bailey. A pseudo-random number generator based on normal numbers, 2004.

[BB12] William C. Barker and Elaine Barker. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block cipher (NIST Special Publication 800-67, revision 1)*. National Institute of Standards and Technology (NIST), January 2012.

[BBDL+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security & Privacy 2015 (Oakland'15)*. IEEE, 2015.

[BBS06] Daniel Brand, Marcio Buss, and Vugranam C. Sreedhar. Evidence-based analysis and inferring preconditions for bug detection (IBM Research report RC24103). Technical report, IBM Research, October 2006.

[BCDS15] Ryad Benadjila, Thomas Calderon, Marion Daubignard, and Markku-Juhani O. Saarinen. CamlCrush: A PKCS#11 filtering proxy. Cryptology ePrint Archive, Report 2015/063, online at `eprint.iacr.org/2015/063`, January 2015. [accessed 2015-01-30].

[BCFS10] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269, Chicago, Illinois, USA, October 2010. ACM Press.

[BDLF+14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, 2014.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. Rapport de recherche RR-7944, INRIA, April 2012.

[BFSW13] Mike Bond, George French, Nigel P. Smart, and Gaven J. Watson. The low-call diet: Authenticated encryption for call counting HSM users. In Ed Dawson, editor, *Topics in Cryptology—CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 359–374. Springer Berlin Heidelberg, 2013.

[BGJ+12] Sergey Bratus, Travis Goodspeed, Peter C. Johnson, Sean W. Smith, and Ryan Speers. Perimeter-crossing buses: a new attack surface for embedded systems. In *Proceedings of the 7th Workshop on Embedded Systems Security (WESS 2012), A Workshop of the Embedded Systems Week (ESWEEK)*, October 2012.

[BGM13] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, ICSE'13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[BJR$^+$14] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (SOSP'14)*, San Jose, CA, May 2014.

[BJS07] Elaine Barker, Don Johnson, and Miles Smid. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (NIST Special Publication 800-56A)*. National Institute of Standards and Technology (NIST), March 2007.

[BK12a] Elaine Barker and John Kelsey. *Recommendation for Random Bit Generator (RBG) Constructions (NIST Special Publication 800-90C, Draft)*. National Institute of Standards and Technology (NIST), August 2012.

[BK12b] Elaine Barker and John Kelsey. *Recommendation for Random Number Generation using Deterministic Random Bit Generators (NIST Special Publication 800-90A)*. National Institute of Standards and Technology (NIST), January 2012.

[BK12c] Elaine Barker and John Kelsey. *Recommendation for the Entropy Sources Used for Random Bit Generation (NIST Special Publication 800-90B, Draft)*. National Institute of Standards and Technology (NIST), August 2012.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. *Lecture Notes in Computer Science*, 1462, 1998.

[BMC$^+$15] Joseph Bonneau, Andrew Miler, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research perspectives and challenges for Bitcoin and cryptocurrencies. Cryptology ePrint Archive, Report 2015/261, online, 2015. `eprint.iacr.org/2015/261` [accessed 2015-04-01].

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, September 2008.

[Bra00] Daniel Brand. A software falsifier (IBM Research report RC21788). Technical report, IBM Research, October 2000.

[CCF$^+$16] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the Tor browser against de-anonymization exploits. In *The annual Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[cd14a] Bitcoin community discussion. (Bitcoin) checkpoint lockin. online at `en.bitcoin.it/wiki/Checkpoint_Lockin`, June 2014. [accessed 2014-08-25].

[cd14b] Bitcoin community discussion. (Bitcoin) transaction confirmation. online at `en.bitcoin.it/wiki/Transaction_confirmation`, May 2014. [accessed 2014-08-26].

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2003.

[Clu03] Jolyon Clulow. On the security of PKCS#11. In *In Proceedings of the 5th International Worshop on Cryptographic Hardware and Embedded Systems (CHES'03), Volume 2779 of LNCS*, pages 411–425. Springer-Verlag, 2003.

[Con07] Jeremy Paul Condit. *Dependent Types for Safe Systems Software*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2007.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[Dic15] Markus Dichtl. Fibonacci ring oscillators as true random number generators - a security risk. Cryptology ePrint Archive, Report 2015/270, online, 2015. `eprint.iacr.org/2015/270` [accessed 2015-04-02].

[DKCC16] Zheng Dong, Kevin Kane, Siyu Chen, and L. Jean Camp. The new wildcats: High-risk banking from worst-case certificate practices online. *Technology Science*, April 2016.

[DKS10] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

[DP10] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 493–504, New York, NY, USA, 2010. ACM.

[Dwo05] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication (NIST Special Publication 800-38B)*. National Institute of Standards and Technology, May 2005.

[EPS15] Chris Evans, Chris Palmer, and Ryan Sleevi. RFC 7469: Public key pinning extension for HTTP, April 2015.

[FG17] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. Cryptology ePrint Archive, Report 2017/082, `eprint.iacr.org/2017/082`, February 2017. [accessed 2017-02-07].

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS'12)*, CCS'12, pages 50–61, New York, NY, USA, 2012. ACM.

[Fil13] Maximilian Johannes Fillinger. Reconstructing the cryptanalytic attack behind the Flame malware. Master's thesis, Universiteit van Amsterdam, September 2013.

[FIP02] National Institute of Standards and Technology. *The Keyed-Hash Message Authentication Code (HMAC)*, March 2002.

[FKD+13] Stephen Farrell, Dirk Kutscher, Christian Dannewitz, Borje Ohlman, Ari Keranen, and Phillip Hallam-Baker. RFC 6920: Naming things with hashes, April 2013.

[FP13] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. *2013 IEEE Symposium on Security and Privacy*, 0:526–540, 2013.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley &amp; Sons, Inc., New York, NY, USA, 2003.

[GBP07] Tobias Gondrom, Ralf Brandner, and Ulrich Pordesch. RFC 4998: Evidence record syntax (ERS), August 2007.

[GHC14] Gasser Gasser, Ralph Holz, and Georg Carle. A deeper understanding of SSH: Results from Internet-wide scans. In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–9. IEEE, 2014.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, October 2012.

[Goo08] Google. Chromium OS design documents, security overview. online at `www.chromium.org/chromium-os/chromiumos-design-docs/security-overview`, 2008. [accessed 2014-02-23].

[Gro97] The Open Group. *fsync - manual page, from The Single UNIX Specification, Version 2*, 1997. [accessed 2015-01-18].

[Gut14] Peter Gutmann. RFC 7366: Encrypt-then-MAC for transport layer security (TLS) and datagram transport layer security (DTLS), September 2014.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[Hea14] OpenSSL "Heartbleed" vulnerability (CVE-2014-0160), April 2014. online at `www.us-cert.gov/ncas/alerts/TA14-098A` [accessed 2015-04-05].

[Hol06] Jason E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 203–211. Australian Computer Society, Inc., 2006.

[Hou04] Russell Housley. RFC 3852: Cryptographic message syntax (CMS), July 2004.

[IEE07] IEEE. IEEE p1619.1, standard for authenticated encryption with length expansion for storage devices, IEEE security in storage working group. *IEEE Std. 1619.1-2007*, December 2007.

[ISO11] ISO/IEC 18031:2011, Information technology, security techniques, random bit generation, 2011.

[JC13] Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting Windows kernel race conditions via memory access patterns. Presented as Black Hat 2013, July 2013.

[JK03] Jakob Jonsson and Burt Kaliski. RFC 3447: Public-key cryptography standards (PKCS)#1: RSA cryptography specifications version 2.1, February 2003.

[KAC12] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. Two Bitcoins at the price of one? double-spending attacks on fast payments in Bitcoin. Cryptology ePrint Archive, Report 2012/248, online, 2012. eprint.iacr.org/2012/248 [accessed 2014-03-02].

[KCR$^+$10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SOSP 2010)*, SOSP'10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[KR02] Vlastimil Klima and Tomas Rosa. Attack on private signature keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP. Cryptology ePrint Archive, Report 2002/076, online, 2002. eprint.iacr.org/2002/076 [accessed 2014-02-18].

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference*, pages 310–331. Springer-Verlag, 2001.

[KS01] Wolfgang Killmann and Werner Schindler. A proposal for: Functionality classes and evaluation methodology for true (physical) random number generators, September 2001.

[KS11] Wolfgang Killmann and Werner Schindler. A proposal for: Functionality classes for random number generators. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), September 2011.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *In Sixth Annual Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999.

[Kub64] Stanley Kubrick. Dr. Strangelove or: How I learned to stop worrying and love the Bomb, 1964.

[KZ14] Anil Kurmus and Robby Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS'14, pages 1366–1377, New York, NY, USA, 2014. ACM.

[Lan14] Adam Langley. Encrypting streams. online at www.imperialviolet.org/2014/06/27/streamingencryption.html, June 2014. [accessed 2014-07-12].

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. RFC 6962: Certificate transparency, June 2013.

[LLS03] Ji Li, Haiyang Liu, and Karen Sollins. Scalable packet classification using bit vector aggregating and folding. MIT LCS Technical memo, MIT/LCS/TM-589, April 2003.

[LM10] Manfred Lochter and Johannes Merkle. RFC 5639: Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation, March 2010.

[LSKL16] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive kernel memory initialization to eliminate data leakages. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 920–932. ACM, 2016.

[MA14] Stephen J. Murdoch and Ross Anderson. Security protocols and evidence: Where many payment systems fail. In *Proceedings of Financial Cryptography and Data Security - 18th International Conference, FC'2014, Barbados*, volume 7859 of *Lecture Notes in Computer Science*. Springer, March 2014.

[Man14] Linux Programmer's Manual. *fsync(2) - Linux manual page*, April 2014. [accessed 2015-01-18].

[MCM06] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, RT'06, pages 46–54, New York, NY, USA, 2006. ACM.

[MKP$^+$95] Barton P. Miller, David Gregory Koski, Cjin Lee Pheow, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report TR-1268, Computer Sciences Department, University of Wisconsin, April 1995.

[Mur08] Steven J. Murdoch. Hardened stateless session cookies. In Bruce Christianson, James A. Malcolm, Vashek Matyas, and Michael Roe, editors, *Security Protocols Workshop*, volume 6615 of *Lecture Notes in Computer Science*, pages 93–101. Springer, 2008.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, March 2009.

[Nat01] National Institute of Standards and Technology. *Advanced Encryption Standard (FIPS 197)*, 2001.

[Nat12] National Institute of Standards and Technology (NIST). *Secure Hash Standard (FIPS 180–4)*, March 2012.

[Nat13] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS) (FIPS 186–4)*, July 2013.

[PBA$^+$05] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Haryadi S. Agrawal, Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the twentieth ACM Symposium on Operating systems principles*, SOSP '05, pages 206–220, New York, NY, USA, 2005. ACM.

[PCA$^+$14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, October 2014.

[PG12] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. *ACM SIGPLAN Notices*, 47(7):215–226, July 2012.

[PHB02] W. Polk, R. Housley, and L. Bassham. RFC 3279: Algorithms and identifiers for the Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, April 2002.

[PKC93] RSA Laboratories. *PKCS 7-Cryptographic Message Syntax Standard (v1.5)*, November 1993.

[PKC04] RSA Laboratories. *PKCS 11–Cryptographic Token Interface Standard (v2.20)*, June 2004.

[PKC15a] OASIS. *PKCS 11-Cryptographic Token Interface Standard Base specification v2.40*, April 2015.

[PKC15b] OASIS. *PKCS 11-Cryptographic Token Interface Standard Current mechanisms specification v2.40*, April 2015.

[Por16] Thomas Pornin. Why constant-time crypto? online at `bearssl.org`/constanttime.html, August 2016. [accessed 2017-01-08].

[PTS$^+$14] Nicolas Palix, Gaël. Thomas, Suman Saha, Cristophe. Calvès, Gilles Muller, and Julia Lawall. Faults in Linux 2.6 (Arxiv 1407.4346). *ArXiv e-prints*, July 2014.

[RBM13] Ohad Rodeh, Josef Bacik, and Chris Mason. btrfs: The Linux B-tree filesystem. *Journal of ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, August 2013.

[RHPFS02] R. R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, April 2002.

[RRDO10] Eric Rescorla, Marsh Ray, Steve Dispensa, and Nasko Oskov. RFC 5746: Transport layer security (TLS) renegotiation indication extension, February 2010.

[RS06] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. Cryptology ePrint Archive, Report 2006/221, 2006. `eprint.iacr.org/2012/221` [accessed 2014-07-02].

[RS12] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. Cryptology ePrint Archive, Report 2012/584, `eprint.iacr.org/2012/584`, 2012. [accessed 2014-07-03].

[SBK$^+$17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albert, and Yarik Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, `eprint.iacr.org/2017/190`, February 2017. [accessed 2017-04-18].

[SCA10] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 485–494, New York, NY, USA, 2010. ACM.

[SEC00] Certicom Research. *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography*, September 2000.

[SEC09] Certicom Research. *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography, version 2.0*, September 2009.

[SF13] Pratik G. Sarkar and Shawn Fitzgerald. Attacks on SSL: a comprehensive study of Beast, CRIME, Time, Breach, Lucky 13 & RC4 biases. Technical report, ISEC Partners, August 2013.

[SKH05] J. Schaad, B. Kaliski, and R. Housley. RFC 4055: Additional algorithms and identifiers for RSA cryptography for use in the internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, June 2005.

[Sma13] Nigel P. Smart. Algorithms, key sizes and parameters report (EU/ENISA) – 2013 recommendations. Technical report, EU/ENISA, October 2013.

[SMZ14] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. BitIodine: extracting intelligence from the Bitcoin network. In *Financial Cryptography and Data Security*, volume (to appear) of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, March 2014.

[ST16] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against Android keystore. Cryptology ePrint Archive, Report 2016/677, `eprint.iacr.org/2016/677`, July 2016. [accessed 2016-08-07].

[Ste14] Graham Steel. Proposal: Authenticated attributes for key wrap in PKCS#11. PKCS#11 standards working group discussion 2014-08-14, August 2014. [accessed 2014-08-26].

[Str16] Falko Strenzke. An analysis of OpenSSL's random number generator. Cryptology ePrint Archive, Report 2016/367, `eprint.iacr.org/2016/367`, April 2016. [accessed 2016-05-23].

[TB10] Sean Turner and Daniel R. L. Brown. RFC 5915: Elliptic curve private key structure, June 2010.

[TBY$^+$09] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk. RFC 5480: Elliptic curve cryptography subject public key information, March 2009.

[Tur10] Sean Turner. RFC 5958: Asymmetric key packages, August 2010.

[VDO14] Tamás Visegrády, Silvio Dragone, and Michael Osborne. Stateless cryptography for virtual environments. *IBM Journal of Research and Development*, 58, January 2014.

[WAP08] Dan Wendlandt, David G. Andersen, and Adrian Perrig. *Perspectives:* improving SSH-style host authentication with multi-path probing. In *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 321–334, June 2008.

[WOW08] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS'08, pages 21–26, New York, NY, USA, 2008. ACM.

[WZKSL13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 260–275, New York, NY, USA, 2013. ACM.

[Zol11] Thierry Zoller. *TLS/SSLv3 renegotiation vulnerability explained*. G-Sec (University of Luxembourg), December 2011.