Document Title	Specification of Crypto Interface for Adaptive Platform
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	883

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-03

Document Change History

Date	Release	Changed by	Description
2018-03-29	18-03	AUTOSAR Release Management	• Crypto API introduced at previous release is renamed to Modeled API, chapter 7 is updated • Added specification of additional Direct Crypto API (chapter 9)
2017-10-27	17-10	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction and functional overview

This specification describes the functionality, API and the configuration for the AUTOSAR Adaptive Crypto Stack as part of the functional cluster Security Management of the AUTOSAR Adaptive Platform.
The Crypto Stack offers applications a standardized interface to cryptographic operations. The Crypto Stack realizes the APIs and manages actual implementations of operations, as well as management functionality handling configuration and brokering.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the Crypto Stack module that are not included in the AUTOSAR glossary [1].

Abbreviation / Acronym:	**Description:**
HSM	Hardware Security Module
TPM	Trusted Platform Module
IPC	Inter-Process Communication

3 Related documentation

3.1 Input documents

[1] Glossary
AUTOSAR_TR_Glossary

[2] Requirements on Security Management for Adaptive Platform
AUTOSAR_RS_SecurityManagement

[3] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

3.2 Related standards and norms

See chapter 3.1.

3.3 Related specification

See chapter 3.1.

4 Constraints and assumptions

4.1 Limitations

The current version of this document is missing some functionality that was available in the AUTOSAR Classic Platform:

- **Secure Counter**
  There is currently no API available to access secure counter primitives that an implementation may provide.

The following functionality is required, but not worked out currently:

- **Asynchronous interface**
  Currently there is only a synchronous API specification and asynchronous behavior must be implemented by the client.
- **Memory management**
  An asynchronous interface requires a specification for managing memory and access to memory (e.g. shared state for `std::shared_ptr` or `std::future`). Currently this has to be addressed by the client.
- **X.509 certificates support**
  Both Crypto APIs don't provide complete support of the X.509 functionality yet. But the Direct Crypto API provides a skeleton of the future X.509 interface that should be extended and completed in the next release.
- **Access Control in the Modeled Crypto API**
  Now modeling and software specification of access control aspects are missed, but should be introduced in the next release.

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other functional clusters

There are currently no dependencies to other functional clusters.

6 Requirements Tracing

The following tables reference the requirements specified in [2] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_CRYPTO_02001]	The Crypto Stack shall conceal symmetric keys from the users	[SWS_CRYPTO_01233] [SWS_CRYPTO_01234] [SWS_CRYPTO_01235] [SWS_CRYPTO_01236] [SWS_CRYPTO_01237] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01240] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01243] [SWS_CRYPTO_20733] [SWS_CRYPTO_23800] [SWS_CRYPTO_23911]
[RS_CRYPTO_02002]	The Crypto Stack shall conceal asymmetric private keys from the users	[SWS_CRYPTO_01233] [SWS_CRYPTO_01234] [SWS_CRYPTO_01235] [SWS_CRYPTO_01236] [SWS_CRYPTO_01237] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01240] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01243] [SWS_CRYPTO_20733] [SWS_CRYPTO_22500] [SWS_CRYPTO_22611]
[RS_CRYPTO_02003]	The Crypto Stack shall support generation and usage of session keys	[SWS_CRYPTO_20512] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_22611] [SWS_CRYPTO_23911]
[RS_CRYPTO_02004]	The Crypto Stack shall support secure storage of different cryptographic artifacts	[SWS_CRYPTO_10006] [SWS_CRYPTO_10600] [SWS_CRYPTO_10603] [SWS_CRYPTO_10611] [SWS_CRYPTO_10618] [SWS_CRYPTO_20517] [SWS_CRYPTO_30112]
[RS_CRYPTO_02005]	The Crypto Stack shall support unique identification of cryptographic objects	[SWS_CRYPTO_10100] [SWS_CRYPTO_20514] [SWS_CRYPTO_20515] [SWS_CRYPTO_30004] [SWS_CRYPTO_30104] [SWS_CRYPTO_30112]

Requirement	Description	Satisfied by
[RS_CRYPTO_02006]	The key objects' identifiers supported by Crypto Stack shall provide information about "origin" and "version" of the key	[SWS_CRYPTO_10100] [SWS_CRYPTO_10111] [SWS_CRYPTO_10112] [SWS_CRYPTO_10113]
[RS_CRYPTO_02007]	The Crypto Stack shall support handling of "secret seeds" in a way similar to secret keys	[SWS_CRYPTO_20723] [SWS_CRYPTO_20734] [SWS_CRYPTO_21311] [SWS_CRYPTO_21411] [SWS_CRYPTO_21514] [SWS_CRYPTO_21515] [SWS_CRYPTO_21811] [SWS_CRYPTO_21812] [SWS_CRYPTO_23000] [SWS_CRYPTO_24015]
[RS_CRYPTO_02008]	The Crypto Stack shall support restrictions of the allowed usage scope for keys and "secret seeds"	[SWS_CRYPTO_10619] [SWS_CRYPTO_21202] [SWS_CRYPTO_21211] [SWS_CRYPTO_21617] [SWS_CRYPTO_23002]
[RS_CRYPTO_02009]	For each key slot the Crypto Stack shall support separation of applications access rights on two categories: owner and users	[SWS_CRYPTO_30002] [SWS_CRYPTO_30003] [SWS_CRYPTO_30117] [SWS_CRYPTO_30121] [SWS_CRYPTO_30122]
[RS_CRYPTO_02101]	The Crypto Stack shall provide interfaces to generate cryptographic keys for all supported primitives	[SWS_CRYPTO_01109] [SWS_CRYPTO_01242] [SWS_CRYPTO_01244] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722]
[RS_CRYPTO_02102]	The Crypto Stack shall prevent keys from being used in incompatible or insecure ways	[SWS_CRYPTO_01235] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01245] [SWS_CRYPTO_01246] [SWS_CRYPTO_10004] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_21212] [SWS_CRYPTO_21213] [SWS_CRYPTO_21412] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21614] [SWS_CRYPTO_21615] [SWS_CRYPTO_21813] [SWS_CRYPTO_21814]

Requirement	Description	Satisfied by
[RS_CRYPTO_02103]	The Crypto Stack shall support primitives to derive cryptographic key material from a base key material	[SWS_CRYPTO_01110] [SWS_CRYPTO_01248] [SWS_CRYPTO_01249] [SWS_CRYPTO_20748] [SWS_CRYPTO_20749] [SWS_CRYPTO_21500] [SWS_CRYPTO_21600]
[RS_CRYPTO_02104]	The Crypto Stack shall support a primitive to exchange cryptographic keys with another entity	[SWS_CRYPTO_01111] [SWS_CRYPTO_01250] [SWS_CRYPTO_01251] [SWS_CRYPTO_01252] [SWS_CRYPTO_20743] [SWS_CRYPTO_20752] [SWS_CRYPTO_20753] [SWS_CRYPTO_20758] [SWS_CRYPTO_21300] [SWS_CRYPTO_21400] [SWS_CRYPTO_21800] [SWS_CRYPTO_24000]
[RS_CRYPTO_02105]	The Crypto Stack shall provide interfaces to import and export cryptographic keys into or from a local key storage or a session objects	[SWS_CRYPTO_01109] [SWS_CRYPTO_01242] [SWS_CRYPTO_01246] [SWS_CRYPTO_01247] [SWS_CRYPTO_20728] [SWS_CRYPTO_20729] [SWS_CRYPTO_20730] [SWS_CRYPTO_20731] [SWS_CRYPTO_20732] [SWS_CRYPTO_23100]
[RS_CRYPTO_02106]	The Crypto Stack shall provide interfaces for secure processing of passwords	[SWS_CRYPTO_20736] [SWS_CRYPTO_20737] [SWS_CRYPTO_20738] [SWS_CRYPTO_22300] [SWS_CRYPTO_22400]
[RS_CRYPTO_02107]	The Crypto Stack shall support the algorithm specification in any key generation or derivation request	[SWS_CRYPTO_10004] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21614] [SWS_CRYPTO_21615]

Requirement	Description	Satisfied by
[RS_CRYPTO_02108]	The Crypto Stack shall provide interfaces for management and usage of algorithm-specific domain parameters	[SWS_CRYPTO_20414] [SWS_CRYPTO_20719] [SWS_CRYPTO_20720] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_20735] [SWS_CRYPTO_20900] [SWS_CRYPTO_21412] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21614] [SWS_CRYPTO_21615] [SWS_CRYPTO_21813] [SWS_CRYPTO_21814] [SWS_CRYPTO_22502] [SWS_CRYPTO_22611] [SWS_CRYPTO_22811] [SWS_CRYPTO_24211] [SWS_CRYPTO_24212] [SWS_CRYPTO_24213] [SWS_CRYPTO_24411]
[RS_CRYPTO_02109]	The Crypto Stack shall support a logically centralized “Key Storage” for all cryptographic artifacts used on the ECU	[SWS_CRYPTO_30000] [SWS_CRYPTO_30100]
[RS_CRYPTO_02110]	The Crypto Stack shall support the prototyping of applications’ access rights and content restrictions of key slot resources	[SWS_CRYPTO_10001] [SWS_CRYPTO_10613] [SWS_CRYPTO_10618] [SWS_CRYPTO_30002] [SWS_CRYPTO_30118]
[RS_CRYPTO_02111]	The Crypto Stack shall provide applications a possibility to define usage restrictions of any new generated or derived key	[SWS_CRYPTO_10005] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21614] [SWS_CRYPTO_21615]
[RS_CRYPTO_02112]	The Crypto Stack shall execute export/import of a key value together with its meta information	[SWS_CRYPTO_20728] [SWS_CRYPTO_20729] [SWS_CRYPTO_20730] [SWS_CRYPTO_20731] [SWS_CRYPTO_20732] [SWS_CRYPTO_23111] [SWS_CRYPTO_23112]

Requirement	Description	Satisfied by
[RS_CRYPTO_02113]	The Crypto Stack interfaces shall support control of the exportability property of a key object	[SWS_CRYPTO_20513] [SWS_CRYPTO_20719] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_20730] [SWS_CRYPTO_20738] [SWS_CRYPTO_21216] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21514] [SWS_CRYPTO_21515] [SWS_CRYPTO_21618]
[RS_CRYPTO_02114]	The Crypto Stack shall support the possibility to restrict the allowed usage of a key individually to each user application	[SWS_CRYPTO_10001] [SWS_CRYPTO_30003] [SWS_CRYPTO_30122]
[RS_CRYPTO_02115]	The Crypto Stack shall enforce assigning required domain parameters to a key in its generation or derivation procedure	[SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_21312] [SWS_CRYPTO_21412] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21614] [SWS_CRYPTO_21813] [SWS_CRYPTO_21814] [SWS_CRYPTO_22502] [SWS_CRYPTO_24016]
[RS_CRYPTO_02116]	The Crypto Stack shall support the version control of key objects kept in the Key Storage according to the key slot prototype	[SWS_CRYPTO_30001] [SWS_CRYPTO_30002]
[RS_CRYPTO_02201]	The Crypto Stack shall provide interfaces to use symmetric encryption and decryption primitives	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207] [SWS_CRYPTO_20200] [SWS_CRYPTO_20742] [SWS_CRYPTO_20744] [SWS_CRYPTO_23600] [SWS_CRYPTO_23700] [SWS_CRYPTO_23900]

Requirement	Description	Satisfied by
[RS_CRYPTO_02202]	The Crypto Stack shall provide interfaces to use asymmetric encryption and decryption primitives	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207] [SWS_CRYPTO_20200] [SWS_CRYPTO_20750] [SWS_CRYPTO_20751] [SWS_CRYPTO_20754] [SWS_CRYPTO_20755] [SWS_CRYPTO_20800] [SWS_CRYPTO_21000] [SWS_CRYPTO_22200] [SWS_CRYPTO_22600] [SWS_CRYPTO_22800] [SWS_CRYPTO_23200]
[RS_CRYPTO_02203]	The Crypto Stack shall provide interfaces to use message authentication code primitives	[SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01217] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01221] [SWS_CRYPTO_01222] [SWS_CRYPTO_01223] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227] [SWS_CRYPTO_01228] [SWS_CRYPTO_20300] [SWS_CRYPTO_20746] [SWS_CRYPTO_22100] [SWS_CRYPTO_23300]

Requirement	Description	Satisfied by
[RS_CRYPTO_02204]	The Crypto Stack shall provide interfaces to use digital signature primitives	[SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01217] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01221] [SWS_CRYPTO_01222] [SWS_CRYPTO_01223] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227] [SWS_CRYPTO_01228] [SWS_CRYPTO_20754] [SWS_CRYPTO_20755] [SWS_CRYPTO_20756] [SWS_CRYPTO_20757] [SWS_CRYPTO_22200] [SWS_CRYPTO_23200] [SWS_CRYPTO_23300] [SWS_CRYPTO_23400] [SWS_CRYPTO_23500] [SWS_CRYPTO_24100]
[RS_CRYPTO_02205]	The Crypto Stack shall provide interfaces to use hashing primitives	[SWS_CRYPTO_01107] [SWS_CRYPTO_01211] [SWS_CRYPTO_01212] [SWS_CRYPTO_01213] [SWS_CRYPTO_01214] [SWS_CRYPTO_01215] [SWS_CRYPTO_01216] [SWS_CRYPTO_20300] [SWS_CRYPTO_20747] [SWS_CRYPTO_21100] [SWS_CRYPTO_23300]
[RS_CRYPTO_02206]	The Crypto Stack shall provide interfaces to configure and use random number generation primitives directly or indirectly by other primitives	[SWS_CRYPTO_01108] [SWS_CRYPTO_01229] [SWS_CRYPTO_01230] [SWS_CRYPTO_01231] [SWS_CRYPTO_20739] [SWS_CRYPTO_20740] [SWS_CRYPTO_20741] [SWS_CRYPTO_22900]
[RS_CRYPTO_02207]	The Crypto Stack shall provide interfaces to use authenticated symmetric encryption and decryption primitives	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207] [SWS_CRYPTO_20100] [SWS_CRYPTO_20745]

Requirement	Description	Satisfied by
[RS_CRYPTO_02208]	The Crypto Stack shall provide interfaces to use symmetric key wrapping primitives	[SWS_CRYPTO_20743] [SWS_CRYPTO_24000]
[RS_CRYPTO_02209]	The Crypto Stack shall provide interfaces to use asymmetric key encapsulation primitives	[SWS_CRYPTO_20752] [SWS_CRYPTO_20753] [SWS_CRYPTO_21400] [SWS_CRYPTO_21800] [SWS_CRYPTO_21900]
[RS_CRYPTO_02301]	The Crypto Stack API shall provide a standardized header files structure	[SWS_CRYPTO_00001] [SWS_CRYPTO_00002] [SWS_CRYPTO_01001] [SWS_CRYPTO_01101] [SWS_CRYPTO_01102] [SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01107] [SWS_CRYPTO_01108] [SWS_CRYPTO_01109] [SWS_CRYPTO_01110] [SWS_CRYPTO_01111] [SWS_CRYPTO_01112] [SWS_CRYPTO_01114] [SWS_CRYPTO_01201] [SWS_CRYPTO_01202] [SWS_CRYPTO_01301] [SWS_CRYPTO_01302]
[RS_CRYPTO_02302]	The Crypto Stack API shall support a streaming approach	[SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01213] [SWS_CRYPTO_01214] [SWS_CRYPTO_01215] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227] [SWS_CRYPTO_20313] [SWS_CRYPTO_20314] [SWS_CRYPTO_20315] [SWS_CRYPTO_20316] [SWS_CRYPTO_20317] [SWS_CRYPTO_20318] [SWS_CRYPTO_23616] [SWS_CRYPTO_23617] [SWS_CRYPTO_23619] [SWS_CRYPTO_23620] [SWS_CRYPTO_23621] [SWS_CRYPTO_23622]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_23623] [SWS_CRYPTO_23624] [SWS_CRYPTO_23711] [SWS_CRYPTO_23712]
[RS_CRYPTO_02304]	The Crypto Stack API should support the possibility to move a state of a “counter mode” stream cipher to a random position	[SWS_CRYPTO_23618]
[RS_CRYPTO_02305]	The Crypto Stack design shall separate cryptographic API from key access API	[SWS_CRYPTO_20700] [SWS_CRYPTO_30100]
[RS_CRYPTO_02306]	The Crypto Stack shall support integration with a Public Key Infrastructure (PKI)	[SWS_CRYPTO_20759] [SWS_CRYPTO_24212] [SWS_CRYPTO_24311] [SWS_CRYPTO_24313] [SWS_CRYPTO_24314] [SWS_CRYPTO_24411] [SWS_CRYPTO_24511] [SWS_CRYPTO_24611] [SWS_CRYPTO_40000] [SWS_CRYPTO_40100] [SWS_CRYPTO_40200] [SWS_CRYPTO_40300] [SWS_CRYPTO_40400] [SWS_CRYPTO_40500] [SWS_CRYPTO_40600]
[RS_CRYPTO_02307]	The Crypto Stack design shall separate cryptographic API from the PKI API	[SWS_CRYPTO_20700] [SWS_CRYPTO_20759] [SWS_CRYPTO_24200] [SWS_CRYPTO_24300] [SWS_CRYPTO_24400] [SWS_CRYPTO_24500] [SWS_CRYPTO_24600]
[RS_CRYPTO_02308]	The Crypto Stack shall support a unified cryptographic primitives naming convention, common for all suppliers	[SWS_CRYPTO_20611] [SWS_CRYPTO_20711] [SWS_CRYPTO_20712]

Requirement	Description	Satisfied by
[RS_CRYPTO_02309]	The Crypto Stack API shall support the run-time configurable usage style	[SWS_CRYPTO_20110] [SWS_CRYPTO_20211] [SWS_CRYPTO_20212] [SWS_CRYPTO_20213] [SWS_CRYPTO_20214] [SWS_CRYPTO_20215] [SWS_CRYPTO_20311] [SWS_CRYPTO_20312] [SWS_CRYPTO_20411] [SWS_CRYPTO_20412] [SWS_CRYPTO_20516] [SWS_CRYPTO_20612] [SWS_CRYPTO_20613] [SWS_CRYPTO_20911] [SWS_CRYPTO_20912] [SWS_CRYPTO_20913] [SWS_CRYPTO_20914] [SWS_CRYPTO_20915] [SWS_CRYPTO_20916] [SWS_CRYPTO_20919] [SWS_CRYPTO_21212] [SWS_CRYPTO_21213] [SWS_CRYPTO_21214] [SWS_CRYPTO_21215] [SWS_CRYPTO_21216] [SWS_CRYPTO_21511] [SWS_CRYPTO_21711] [SWS_CRYPTO_21712] [SWS_CRYPTO_21713] [SWS_CRYPTO_21714] [SWS_CRYPTO_21911] [SWS_CRYPTO_21912] [SWS_CRYPTO_22011] [SWS_CRYPTO_22411] [SWS_CRYPTO_22412] [SWS_CRYPTO_23411] [SWS_CRYPTO_23412] [SWS_CRYPTO_23413] [SWS_CRYPTO_23611] [SWS_CRYPTO_23612] [SWS_CRYPTO_23613] [SWS_CRYPTO_23614] [SWS_CRYPTO_23615] [SWS_CRYPTO_23711] [SWS_CRYPTO_23912]

Requirement	Description	Satisfied by
[RS_CRYPTO_02310]	The Crypto Stack API shall support an efficient mechanism of error states notification	[SWS_CRYPTO_19000] [SWS_CRYPTO_19100] [SWS_CRYPTO_19111] [SWS_CRYPTO_19112] [SWS_CRYPTO_19113] [SWS_CRYPTO_19600] [SWS_CRYPTO_19700] [SWS_CRYPTO_19800] [SWS_CRYPTO_19900] [SWS_CRYPTO_29400] [SWS_CRYPTO_29500] [SWS_CRYPTO_29600] [SWS_CRYPTO_29700] [SWS_CRYPTO_29800] [SWS_CRYPTO_29900] [SWS_CRYPTO_39900] [SWS_CRYPTO_49500] [SWS_CRYPTO_49600] [SWS_CRYPTO_49700] [SWS_CRYPTO_49800] [SWS_CRYPTO_49900]
[RS_CRYPTO_02311]	The Crypto Stack API specification should be complete and allow flexible usage of the stack functionality	[SWS_CRYPTO_10110] [SWS_CRYPTO_10210] [SWS_CRYPTO_10311] [SWS_CRYPTO_10312] [SWS_CRYPTO_10411] [SWS_CRYPTO_10610] [SWS_CRYPTO_10612] [SWS_CRYPTO_10614] [SWS_CRYPTO_10615] [SWS_CRYPTO_10616] [SWS_CRYPTO_10617] [SWS_CRYPTO_19101] [SWS_CRYPTO_19110] [SWS_CRYPTO_20001] [SWS_CRYPTO_20002] [SWS_CRYPTO_20003] [SWS_CRYPTO_20210] [SWS_CRYPTO_20216] [SWS_CRYPTO_20217] [SWS_CRYPTO_20310] [SWS_CRYPTO_20400] [SWS_CRYPTO_20413] [SWS_CRYPTO_20500] [SWS_CRYPTO_20503]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_20511] [SWS_CRYPTO_20600] [SWS_CRYPTO_20601] [SWS_CRYPTO_20602] [SWS_CRYPTO_20701] [SWS_CRYPTO_20702] [SWS_CRYPTO_20710] [SWS_CRYPTO_20901] [SWS_CRYPTO_20902] [SWS_CRYPTO_20917] [SWS_CRYPTO_20918] [SWS_CRYPTO_21611] [SWS_CRYPTO_21612] [SWS_CRYPTO_21613] [SWS_CRYPTO_21616] [SWS_CRYPTO_21700] [SWS_CRYPTO_21910] [SWS_CRYPTO_22000] [SWS_CRYPTO_22311] [SWS_CRYPTO_22312] [SWS_CRYPTO_22313] [SWS_CRYPTO_22314] [SWS_CRYPTO_22315] [SWS_CRYPTO_22316] [SWS_CRYPTO_22317] [SWS_CRYPTO_22318] [SWS_CRYPTO_22320] [SWS_CRYPTO_22711] [SWS_CRYPTO_22712] [SWS_CRYPTO_22713] [SWS_CRYPTO_22901] [SWS_CRYPTO_22911] [SWS_CRYPTO_22912] [SWS_CRYPTO_22913] [SWS_CRYPTO_23101] [SWS_CRYPTO_23110] [SWS_CRYPTO_23310] [SWS_CRYPTO_23311] [SWS_CRYPTO_23410] [SWS_CRYPTO_24011] [SWS_CRYPTO_24012] [SWS_CRYPTO_24013] [SWS_CRYPTO_24014] [SWS_CRYPTO_24111] [SWS_CRYPTO_24112] [SWS_CRYPTO_24312] [SWS_CRYPTO_24412] [SWS_CRYPTO_30101]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_30103] [SWS_CRYPTO_30106] [SWS_CRYPTO_30110] [SWS_CRYPTO_30113] [SWS_CRYPTO_30114] [SWS_CRYPTO_30115] [SWS_CRYPTO_30116] [SWS_CRYPTO_30119] [SWS_CRYPTO_40101] [SWS_CRYPTO_40111] [SWS_CRYPTO_40112] [SWS_CRYPTO_40113] [SWS_CRYPTO_40114] [SWS_CRYPTO_40115] [SWS_CRYPTO_40211] [SWS_CRYPTO_40212] [SWS_CRYPTO_40213] [SWS_CRYPTO_40214] [SWS_CRYPTO_40215] [SWS_CRYPTO_40216] [SWS_CRYPTO_40217] [SWS_CRYPTO_40218] [SWS_CRYPTO_40219] [SWS_CRYPTO_40311] [SWS_CRYPTO_40312] [SWS_CRYPTO_40402] [SWS_CRYPTO_40410] [SWS_CRYPTO_40411] [SWS_CRYPTO_40412] [SWS_CRYPTO_40413] [SWS_CRYPTO_40414] [SWS_CRYPTO_40415] [SWS_CRYPTO_40416] [SWS_CRYPTO_40510] [SWS_CRYPTO_40511] [SWS_CRYPTO_40601]
[RS_CRYPTO_02401]	The Crypto Stack should support a joint usage of multiple back-end cryptography providers including ones with non-extractable keys	[SWS_CRYPTO_01208] [SWS_CRYPTO_01209] [SWS_CRYPTO_01210] [SWS_CRYPTO_01303] [SWS_CRYPTO_01304] [SWS_CRYPTO_01305] [SWS_CRYPTO_01306] [SWS_CRYPTO_01307] [SWS_CRYPTO_01308] [SWS_CRYPTO_01309] [SWS_CRYPTO_01310] [SWS_CRYPTO_01311] [SWS_CRYPTO_10002] [SWS_CRYPTO_20000] [SWS_CRYPTO_20614] [SWS_CRYPTO_20700] [SWS_CRYPTO_30000] [SWS_CRYPTO_30100] [SWS_CRYPTO_30120]

Requirement	Description	Satisfied by
[RS_CRYPTO_02402]	The Crypto Stack shall support prioritizing the processing of requests	[SWS_CRYPTO_01232]
[RS_CRYPTO_02403]	The Crypto Stack shall support isolating keys and requests	[SWS_CRYPTO_01243] [SWS_CRYPTO_21200] [SWS_CRYPTO_22500] [SWS_CRYPTO_22700] [SWS_CRYPTO_23800]
[RS_CRYPTO_02404]	Design of the Crypto Stack interfaces shall support minimization of resources consumption	[SWS_CRYPTO_01236] [SWS_CRYPTO_01240] [SWS_CRYPTO_01245] [SWS_CRYPTO_01246] [SWS_CRYPTO_10001] [SWS_CRYPTO_10002] [SWS_CRYPTO_10003] [SWS_CRYPTO_10004] [SWS_CRYPTO_10005] [SWS_CRYPTO_10100] [SWS_CRYPTO_10200] [SWS_CRYPTO_10211] [SWS_CRYPTO_10300] [SWS_CRYPTO_10400] [SWS_CRYPTO_10601] [SWS_CRYPTO_10602] [SWS_CRYPTO_20101] [SWS_CRYPTO_20312] [SWS_CRYPTO_20318] [SWS_CRYPTO_20319] [SWS_CRYPTO_20320] [SWS_CRYPTO_20321] [SWS_CRYPTO_20501] [SWS_CRYPTO_20502]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_20703] [SWS_CRYPTO_20704] [SWS_CRYPTO_20705] [SWS_CRYPTO_20706] [SWS_CRYPTO_20707] [SWS_CRYPTO_20713] [SWS_CRYPTO_20714] [SWS_CRYPTO_20715] [SWS_CRYPTO_20716] [SWS_CRYPTO_20719] [SWS_CRYPTO_20720] [SWS_CRYPTO_20721] [SWS_CRYPTO_20722] [SWS_CRYPTO_20723] [SWS_CRYPTO_20724] [SWS_CRYPTO_20725] [SWS_CRYPTO_20726] [SWS_CRYPTO_20727] [SWS_CRYPTO_20733] [SWS_CRYPTO_20734] [SWS_CRYPTO_20735] [SWS_CRYPTO_20736] [SWS_CRYPTO_20737] [SWS_CRYPTO_20738] [SWS_CRYPTO_20739] [SWS_CRYPTO_20740] [SWS_CRYPTO_20741] [SWS_CRYPTO_20742] [SWS_CRYPTO_20743] [SWS_CRYPTO_20744] [SWS_CRYPTO_20745] [SWS_CRYPTO_20746] [SWS_CRYPTO_20747] [SWS_CRYPTO_20748] [SWS_CRYPTO_20749] [SWS_CRYPTO_20750] [SWS_CRYPTO_20751] [SWS_CRYPTO_20752] [SWS_CRYPTO_20753] [SWS_CRYPTO_20754] [SWS_CRYPTO_20755] [SWS_CRYPTO_20756] [SWS_CRYPTO_20757] [SWS_CRYPTO_20758] [SWS_CRYPTO_20759] [SWS_CRYPTO_20801] [SWS_CRYPTO_21001] [SWS_CRYPTO_21101]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_21201] [SWS_CRYPTO_21301] [SWS_CRYPTO_21311] [SWS_CRYPTO_21312] [SWS_CRYPTO_21400] [SWS_CRYPTO_21401] [SWS_CRYPTO_21411] [SWS_CRYPTO_21412] [SWS_CRYPTO_21501] [SWS_CRYPTO_21512] [SWS_CRYPTO_21513] [SWS_CRYPTO_21514] [SWS_CRYPTO_21515] [SWS_CRYPTO_21601] [SWS_CRYPTO_21618] [SWS_CRYPTO_21800] [SWS_CRYPTO_21801] [SWS_CRYPTO_21811] [SWS_CRYPTO_21812] [SWS_CRYPTO_21813] [SWS_CRYPTO_21814] [SWS_CRYPTO_22101] [SWS_CRYPTO_22201] [SWS_CRYPTO_22301] [SWS_CRYPTO_22319] [SWS_CRYPTO_22401] [SWS_CRYPTO_22501] [SWS_CRYPTO_22502] [SWS_CRYPTO_22701] [SWS_CRYPTO_23001] [SWS_CRYPTO_23201] [SWS_CRYPTO_23301] [SWS_CRYPTO_23501] [SWS_CRYPTO_23511] [SWS_CRYPTO_23512] [SWS_CRYPTO_23513] [SWS_CRYPTO_23601] [SWS_CRYPTO_23701] [SWS_CRYPTO_23801] [SWS_CRYPTO_24000] [SWS_CRYPTO_24001] [SWS_CRYPTO_24015] [SWS_CRYPTO_24016] [SWS_CRYPTO_24101] [SWS_CRYPTO_24201] [SWS_CRYPTO_24212] [SWS_CRYPTO_24301] [SWS_CRYPTO_24401]

Requirement	Description	Satisfied by
		[SWS_CRYPTO_24411] [SWS_CRYPTO_24501] [SWS_CRYPTO_24511] [SWS_CRYPTO_24601] [SWS_CRYPTO_24611] [SWS_CRYPTO_30002] [SWS_CRYPTO_30003] [SWS_CRYPTO_30004] [SWS_CRYPTO_30102] [SWS_CRYPTO_30105] [SWS_CRYPTO_40201] [SWS_CRYPTO_40301] [SWS_CRYPTO_40401] [SWS_CRYPTO_40501]
[RS_CRYPTO_02405]	The Crypto Stack shall support the key slots identification in a way independent from a concrete deployment	[SWS_CRYPTO_10003] [SWS_CRYPTO_30105] [SWS_CRYPTO_30111]
[RS_CRYPTO_02406]	The Crypto Stack shall support its efficient usage in the “real-time” and “non-real-time” modes	[SWS_CRYPTO_20717] [SWS_CRYPTO_20718]

7 Functional specification

The AUTOSAR Adaptive architecture organizes the software of the AUTOSAR Adaptive foundation as functional clusters. These clusters offer common functionality as services to the applications. The Security Management (SEC) for AUTOSAR Adaptive is such a functional cluster and is part of “AUTOSAR Runtime for Adaptive Applications” - ARA. The functional cluster consists of multiple modules. The Crypto Stack is a module of this functional cluster that offers interfaces to Adaptive applications. It is responsible for the construction and supervision of cryptographic primitives.

The Crypto Stack provides the infrastructure to access multiple implementations of cryptographic algorithms through a standardized interface.

This specification includes the syntax of the API, the relationship of the API to the model and describes semantics. The specification does not pose constraints on the internal architecture and implementation of the Crypto Stack.

7.1 Architectural concepts

The Crypto Stack of AUTOSAR Adaptive can be logically divided into the following parts:

- Language binding
- Drivers
- Crypto Stack management software.

There are several types of interfaces available in the context of the Crypto Stack:

- **Public Interface**
  Part of the AUTOSAR Adaptive API and specified in this document. This is the standardized API presented in the name space `ara::crypto`. It includes a few sub-domain APIs (`common, crypto, keys, x509`).
- **Protected Interface**
  Used for interaction between functional clusters. This may be a custom API, but it can also re-use the Public Interface.
- **Private Interface**
  Used for interaction within the module. These interfaces are not described in the specification and are implementation-specific.

The AUTOSAR Adaptive Crypto Stack supports two different API:

- **Modeled API** is a simplified API, available via `ClientServerInterface`s according to the modeled `CryptoJob`s.
- **Direct API** is a flexible API, available directly via linked library.

For the design of the both ARA Crypto APIs the following constraints were applied:

- Support the independence of application software components from a specific platform implementation.
- Make the API as lean as possible, no specific use cases are supported which could also be layered on top of the API.
- Offer a “comfort layer” to enable the use of C++11/14 features.
- Support the integration into safety relevant systems.

Therefore the API of the Crypto Stack follows a specific set of design decisions:

- It uses a pure virtual API to access different algorithms through a unified interface.
- The memory management controllable by the caller.
- Its API has zero-copy capabilities.
- A “comfort layer” provides functionality like asynchronous operation (not implemented yet).

Differences of the APIs designs:

- **Modeled API** makes the API as lean as possible, no specific use cases are supported which could also be layered on top of the API.
- **Direct API** provides maximal flexibility to enable efficient usage of the cryptography by `Adaptive Application`s and their portability between different platform suppliers.

7.2 Integration of Adaptive Application and Crypto Stack

The `Adaptive Application` should not have direct access to keys within its own process. Therefore the Crypto Stack has to support features for isolating Adaptive Applications from the Crypto Stack implementation. The following separation mechanisms are envisioned:

1. Process isolation
2. Hardware isolation

The two mechanisms will be outlined briefly below.

7.2.1 Process isolation

The implementer/integrator of this specification may choose to isolate the cryptographic algorithm implementation from the `Adaptive Application`s by means of separating them into two different processes. Generally it is strongly recommended to separate the Crypto Stack on two domains: “front-end” and isolated “back-end”. At least the Crypto Stack back-end shall include all functionality that is subject of access control and implementations of cryptographic transformations that handle secret/private key material. It should be done in following ways:

- **Modeled API.** The Crypto Stack implementation shall provide ClientServerInterfaces according to the modeled CryptoJobs. The connection between the two pieces of software is implicitly made by the mapping of CryptoNeed onto CryptoJob. The communication interface is described by the ClientServerInterface. The interface visible to the Adaptive Application developer is specified in section 8.1.2.1. The actual IPC protocol is specific per platform implementation.
- **Direct API.** The Crypto Stack back-end can be implemented in form of privileged independent process (e.g. driver or daemon). But independently from details of the Crypto Stack back-end implementation, its front-end should be presented to Adaptive Application’s developers in form of a library, which hides all implementation and communication details. The front-end library can directly implement key-less transformations (like hashing) and public key transformations (like signature verification). The interface visible to the Adaptive Applications’ developers is specified in section 9.

7.2.2 Hardware isolation

The implementer/integrator of this specification may choose to isolate the cryptographic algorithm implementation from the Adaptive Applications by means of separating them using a hardware mechanism (e.g. HSM, TPM). It should be done in following ways:

- **Modeled API.** The Crypto Stack implementation shall provide CryptoJobs that are implemented in the selected hardware. The connection between the software and hardware is made by the mapping of CryptoNeed onto CryptoJobs that can be identifier in the hardware or its driver. The interface visible to the Adaptive Application developer is specified in section 8.1.2.1. The actual implementation of the driver is specific per platform implementation.
- **Direct API.** All communications with hardware should be implemented in the Crypto Stack “back-end” (in form of a driver), all other aspects are similar to the section 7.2.1. Switching between a hardware and a software based implementations is fully transparent for Adaptive Application developers, in both cases they use the API specified in section 9.

7.3 Supported algorithms

At least the following cryptographic algorithms or primitives should be supported by the Crypto Stack:

- Random Number Generation
  - Deterministic Random Number Generator (DRNG)
  - True Random Number Generator (TRNG)
- Symmetric Ciphers
  - AES
    - Key Length: 128, 192 and 256 bits
    - Modes: ECB, OFB, CFB, CBC, CTR, GCM, CCM, Poly1305
  - Camellia
    - Key Length: 128, 192 and 256 bits
    - Modes: ECB, OFB, CFB, CBC, CTR, GCM, CCM
  - ChaCha20
- Asymmetric Encryption/Decryption and Signature Handling
  - RSA
    - Key Length: 2048, 3072 and 4096 bits
    - Padding: PKCS#1 v2.2
  - Curve25519/Ed25519
  - NIST curves P256, P384 and P521 / ECDSA
- Hash
  - SHA-1
  - SHA-2
    - Length: 256, 384 and 512 bits
  - SHA-3
    - Length: 256, 384 and 512 bits
- Message Authentication Code (MAC)
  - CBC-MAC
  - CMAC
  - GMAC

    - HMAC
- Key Agreement
    - Diffie-Hellman (DH)
    - Elliptic Curve DH (ECDH)
- Key Derivation Function
    - HKDF
    - PBKDF1
    - PBKDF2
- Key Encapsulation Mechanism
    - RSA-KEM
    - ECIES-KEM
    - PSEC-KEM
    - ACE-KEM

The Crypto Stack should support handling the following cryptographic objects:

- Certificate Management
    - Handling of X.509 Certificates
    - Im/Export in DER format
    - Creation of CSRs

8 Modeled API specification

The API supports a streaming interface and a single call interface. Selected interfaces therefore provide the following methods:

- `Start`
- `Update`
- `Finish`
- `Process`

The `Start` method resets the internal states of the algorithm to begin processing chunks of data. The `Update` method updates the internal state of the algorithm by processing the given chunk of data and, if feasible, returning a transformed data chunk. The `Finish` method concludes the operation cycle of the algorithm by returning the final transformation.
The `Process` method can be used if all data is available at once and shall be processed in a single call. The `Process` method may internally call `Start`, `Update` and `Finish`.

8.1 Modeled API C++ language binding

8.1.1 API Header files

This chapter describes the header files of the ara::crypto API. The input for the header files are AUTOSAR Adaptive meta model classes within the `CryptoNeed` description, as defined in the AUTOSAR Manifest Specification [3].

The following requirements are applicable for all header files.

[SWS_CRYPTO_00001] No memory allocation in header files ⌈ The header files shall not contain code that creates objects on the heap. ⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_00002] Folder structure ⌈ The *CryptoNeed header files* defined by [SWS_CRYPTO_01001] and the *Common header file* defined by [SWS_CRYPTO_01101], [SWS_CRYPTO_01103], [SWS_CRYPTO_01104], [SWS_CRYPTO_01105], [SWS_CRYPTO_01106], [SWS_CRYPTO_01107], [SWS_CRYPTO_01108], [SWS_CRYPTO_01109], [SWS_CRYPTO_01110], [SWS_CRYPTO_01111], [SWS_CRYPTO_01112] and [SWS_CRYPTO_01113] shall be located within the folder:

```
<folder>/
```

where:
`<folder>` is the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

8.1.1.1 CryptoNeed header files

The *CryptoNeed header files* are the central definitions of the ara::crypto API that are required to perform cryptographic operations.

[SWS_CRYPTO_01001] CryptoNeed header files existence ⌈ The Crypto Stack shall provide one *CryptoNeed header file* for each `CryptoNeed` defined in the modeling and mapped onto the Adaptive Application's `PortPrototype`'s. The file name for the *CryptoNeed header file* shall be `<name>_cryptoneed.h`, where `<name>` is the `CryptoNeed.shortName` converted to lower-case letters. ⌋*(RS_CRYPTO_02301)*

8.1.1.2 Common header files

The *Common header files* are central definitions of the ara::crypto API that are required to describe the API of the Crypto Stack.

[SWS_CRYPTO_01101] Existence of Span header file ⌈ The Crypto Stack shall provide the *Common header file* `span.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

The `Span` defines a container class that has non-owning properties while still offering the benefits of iterators and size information. The API is shown in the appendix' section B.

[SWS_CRYPTO_01102] An ara::crypto::Span header file shall be provided by the implementation ⌈ The Crypto Stack shall provide a `ara::crypto::Span` implementation in the *Common header file* defined in [SWS_CRYPTO_01101]. The implementation shall be aliased as follows:

```
namespace ara {
namespace crypto {

using Span = span;

}
}
```

⌋*(RS_CRYPTO_02301)*

The Crypto Stack supports standardized access to selected primitives. The standardized access is ensured by the definition of pure virtual C++ interfaces as part of the *Common header files*. The API for these interfaces is shown in section 8.1.2.

[SWS_CRYPTO_01103] A cipher interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `cipher.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

[SWS_CRYPTO_01104] A cipher parameters interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `cipher_parameters.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01105] A signer interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `signer.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01106] A verifier interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `verifier.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01107] A hash interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `hash.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02205)*

[SWS_CRYPTO_01108] A random number generation interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `random.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02206)*

[SWS_CRYPTO_01109] A key management interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_management.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02105, RS_CRYPTO_02101)*

[SWS_CRYPTO_01110] A key derivation interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_derivation.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02103)*

[SWS_CRYPTO_01111] A key exchange interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_exchange.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02104)*

[SWS_CRYPTO_01112] A key interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01113] A keyed primitive interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `keyed_primitive.h`. The file shall be located in the start folder for the ara::crypto header files specific for a project or platform vendor. ⌋*()*

[SWS_CRYPTO_01114] Every header file shall include the ara::crypto::Span header file ⌈ The *Common header files* shall include the *Common header file* defined in [SWS_CRYPTO_01101]:

```
#include "ara/crypto/span.h"
```

⌋*(RS_CRYPTO_02301)*

8.1.2 API Reference

8.1.2.1 Common API

The *Common header files* have a static interface that is described in the *Common API*.

8.1.2.1.1 KeyedPrimitive interface

[SWS_CRYPTO_01201] Keyed primitive interface ⌈ The Crypto Stack shall provide a pure virtual interface for `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01113] in the namespace `ara::crypto`.

```
class KeyedPrimitive
```

⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01202] GetKey method ⌈ The Crypto Stack shall provide a method to retrieve a representation of a key associated with a primitive for the `KeyedPrimitive` interface.

```
virtual Key const& GetKey() const = 0;
```

⌋*(RS_CRYPTO_02301)*

The returned `Key` should not contain raw key data but should be a proxy to query information about the key or to use it in another primitive.

8.1.2.1.2 Cipher interface

[SWS_CRYPTO_01203] Cipher interface ⌈ The Crypto Stack shall provide an interface for `Cipher` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01103] in the namespace `ara::crypto`.

```
class Cipher : public KeyedPrimitive
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

The Cipher interface is employed for encrypting and decrypting data. The transformation may be done by a symmetric or asymmetric algorithm. The Cipher interface supports a single call interface via the Process method or a streaming interface via the Start, Update and Finish methods.

[SWS_CRYPTO_01204] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the cipher. The start methods accepts an optional argument of the type CipherParameters to initialize the cipher if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)* **[SWS_CRYPTO_01205] Update method** ⌈ The Crypto Stack shall provide a method to update an operation cycle of the cipher. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the input variable and the transformed data will be supplied in the output variable.

```
virtual void Update(ara::crypto::Span<const uint8_t> input, ara::crypto
    ::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)*

[SWS_CRYPTO_01206] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the cipher. The method closes the operation of the cipher and may return final bytes from the cipher operation in the variable output.

```
virtual void Finish(ara::crypto::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)*

[SWS_CRYPTO_01207] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The Process method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01204], [SWS_CRYPTO_01205] and [SWS_CRYPTO_01206].

```
virtual void Process(CipherParameters* parameters, ara::crypto::Span<
    const uint8_t> input, ara::crypto::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

8.1.2.1.3 CipherParameters interface

[SWS_CRYPTO_01208] CipherParameters interface ⌈ The Crypto Stack shall provide an interface for CipherParameters located in the header file defined in [SWS_CRYPTO_01104] in the namespace ara::crypto.

```
class CipherParameters
```

⌋(*RS_CRYPTO_02401*)

[SWS_CRYPTO_01209] GetFlags method ⌈ The Crypto Stack shall provide a method to query the flags for optimizing the algorithms operation. The returned value may be used by the implementation to select implementation strategies.

```
virtual AlgorithmFlags GetFlags() const = 0;
```

⌋(*RS_CRYPTO_02401*)

[SWS_CRYPTO_01210] GetNonceIV method ⌈ The Crypto Stack shall provide a method to retrieve a nonce or initialization vector for an algorithm to use.

```
virtual ara::crypto::Span<uint8_t> const GetNonceIV() const = 0;
```

⌋(*RS_CRYPTO_02401*)

[SWS_CRYPTO_01232] AlgorithmFlags enumeration ⌈ The Crypto Stack shall provide an enumeration of algorithm flags. The following flags shall be supported:

```
enum class AlgorithmFlags
{
  None,
  Latency,
  Background
};
```

The `AlgorithmFlags` values' semantics are described in table 8.1. ⌋ (*RS_CRYPTO_02402*)

AlgorithmFlag	**Explanation**
`None`	No optimization preferences.
`Latency`	Optimize for latency.
`Background`	Optimize for processing in the background.

Table 8.1: AlgorithmFlags

8.1.2.1.4 Hash interface

[SWS_CRYPTO_01211] Hash interface ⌈ The Crypto Stack shall provide an interface for `Hash` located in the header file defined in [SWS_CRYPTO_01107] in the namespace `ara::crypto`.

```
class Hash
```

⌋(*RS_CRYPTO_02205*)

[SWS_CRYPTO_01212] GetDigestSize method ⌈ The Crypto Stack shall provide a method to query the size of the produced digest. The size shall be returned in bytes.

```
virtual AlgorithmFlags GetFlags() const = 0;
```

⌋(RS_CRYPTO_02205)

[SWS_CRYPTO_01213] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the hash.

```
virtual void Start() = 0;
```

⌋(RS_CRYPTO_02205, RS_CRYPTO_02302)

[SWS_CRYPTO_01214] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the hash. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable.

```
virtual void Update(ara::crypto::Span<const uint8_t> input) = 0;
```

⌋(RS_CRYPTO_02205, RS_CRYPTO_02302)

[SWS_CRYPTO_01215] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the hash. The method closes the operation of the hash and return the digest in the variable `digest`.

```
virtual void Finish(ara::crypto::Span<uint8_t> digest) = 0;
```

⌋(RS_CRYPTO_02205, RS_CRYPTO_02302)

[SWS_CRYPTO_01216] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the hash in a single call. The `Process` method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01214] and [SWS_CRYPTO_01215].

```
virtual void Process(ara::crypto::Span<const uint8_t> input, ara::
    crypto::Span<uint8_t> digest) = 0;
```

⌋(RS_CRYPTO_02205)

8.1.2.1.5 Signer interface

[SWS_CRYPTO_01217] Signer interface ⌈ The Crypto Stack shall provide an interface for `Signer` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01105] in the namespace `ara::crypto`.

```
class Signer : public KeyedPrimitive
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

The `Signer` interface is employed for signing data. The signature may be created using symmetric or asymmetric algorithms. The `Signer` interface supports a single call interface via the `Process` method or a streaming interface via the `Start`, `Update` and `Finish` methods.

[SWS_CRYPTO_01222] GetTagSize method ⌈ The Crypto Stack shall provide a method to query the size of the produced signature. The size shall be reported in bytes.

```
virtual std::size_t GetTagSize() const = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

[SWS_CRYPTO_01218] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the signer. The start methods accepts an optional argument of the type CipherParameters to initialize the signer if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)
[SWS_CRYPTO_01219] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the signer. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the input variable.

```
virtual void Update(ara::crypto::Span<const uint8_t> input) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01220] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the signer. The method closes the operation of the signer and returns the signature bytes in the variable output.

```
virtual void Finish(ara::crypto::Span<uint8_t> output) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01221] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The Process method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01218], [SWS_CRYPTO_01219] and [SWS_CRYPTO_01220].

```
virtual void Process(CipherParameters* parameters, ara::crypto::Span<
    const uint8_t> input, ara::crypto::Span<uint8_t> output) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

8.1.2.1.6 Verifier interface

[SWS_CRYPTO_01223] Verifier interface ⌈ The Crypto Stack shall provide an interface for Verifier derived from KeyedPrimitive located in the header file defined in [SWS_CRYPTO_01106] in the namespace ara::crypto.

```
class Verifier : public KeyedPrimitive
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

The Verifier interface is employed for verifying the authenticity data. The signature may be created using symmetric or asymmetric algorithms. The Verifier interface supports a single call interface via the Process method or a streaming interface via the Start, Update and Finish methods.

[SWS_CRYPTO_01225] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the verifier. The start methods accepts an optional argument of the type `CipherParameters` to initialize the verifier if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01226] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the verifier. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable.

```
virtual void Update(ara::crypto::Span<const uint8_t> input) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01227] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the verifier. The method closes the operation of the verifier and performs the verification. The original authenticator is given in the variable `authenticator` and will be compared against the computed one. The optional argument `length` describes the amount of leftmost bits that shall be considered in the comparison, it is therefore given in bits. If its not present the entire authenticator is relevant.
If the relevant bits of the given `authenticator` match the computed authenticator `true` is returned, `false` otherwise.

```
virtual bool Finish(ara::crypto::Span<const uint8_t> authenticator, std
    ::size_t length = 0) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01228] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The `Process` method uses the same parameters with identical semantics and the same return values semantics as described for the streaming interface in [SWS_CRYPTO_01225], [SWS_CRYPTO_01226] and [SWS_CRYPTO_01227].

```
virtual bool Process(CipherParameters* parameters, ara::crypto::Span<
    const uint8_t> input, ara::crypto::Span<const uint8_t>
    authenticator, std::size_t length = 0) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

8.1.2.1.7 Random number generation interface

[SWS_CRYPTO_01229] Random interface ⌈ The Crypto Stack shall provide an interface for `Random` located in the header file defined in [SWS_CRYPTO_01108] in the namespace `ara::crypto`.

```
class Random
```

⌋(RS_CRYPTO_02206)

[SWS_CRYPTO_01230] Seed method ⌈ The Crypto Stack shall provide a method to add additional random data to increase entropy of the random number generator. The additional entropy can be provided in the variable `input`.

```
virtual void Seed(ara::crypto::Span const input);
```

⌋*(RS_CRYPTO_02206)*

[SWS_CRYPTO_01231] Generate method ⌈ The Crypto Stack shall provide a method to generate random data. The random data will be place in the variable `output`.

```
virtual void Generate(ara::crypto::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02206)*

8.1.2.1.8 Key interface

[SWS_CRYPTO_01233] Key interface ⌈ The Crypto Stack shall provide an interface for `Key` located in the header file defined in [SWS_CRYPTO_01112] in the namespace `ara::crypto`.

```
class Key
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002)*

[SWS_CRYPTO_01234] GetId method ⌈ The Crypto Stack shall provide a method to query the unique identification of the key.

```
virtual uint32_t GetId() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002)*

[SWS_CRYPTO_01235] GetUsage method ⌈ The Crypto Stack shall provide a method to query the allowed usages of the key. See table 8.2 for more details.

```
virtual KeyUsage GetUsage() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

[SWS_CRYPTO_01236] GetProtection method ⌈ The Crypto Stack shall provide a method to query the protection flags of the key. See table 8.3 for more details.

```
virtual KeyProtection GetProtection() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02404)*

[SWS_CRYPTO_01237] GetSize method ⌈ The Crypto Stack shall provide a method to query the size of the key. The size shall be returned in bits.

```
virtual std::size_t GetSize() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002)*

[SWS_CRYPTO_01238] GetType method ⌈ The Crypto Stack shall provide a method to query the type of the key. See table 8.4 for more details.

```
virtual KeyType GetType() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

[SWS_CRYPTO_01239] KeyUsage enumeration ⌈The Crypto Stack shall provide an enumeration of key usage flags. The following flags shall be supported:

```
#define CKI_BIT(n) (1 << (n))

enum class KeyUsage : uint32_t
{
  Encrypt     = CKI_BIT(1),
  Decrypt     = CKI_BIT(2),
  Sign        = CKI_BIT(3),
  Verify      = CKI_BIT(4),
  Exchange    = CKI_BIT(5),
  Derive      = CKI_BIT(6),
  Provision   = CKI_BIT(7),
  Migration   = CKI_BIT(8)
};
```

The `KeyUsage` values' semantics are described in table 8.2. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

KeyUsage	**Explanation**
`Encrypt`	The key may be used for encryption.
`Decrypt`	The key may be used for decryption.
`Sign`	The key may be used for signing.
`Verify`	The key may be used for verification.
`Exchange`	The key may be used for key exchange.
`Derive`	The key may be used as a base key for deriving other keys.
`Provision`	The key may be used for unwrapping keys during key provisioning.
`Migration`	The key may be used for exporting keys for migration.

Table 8.2: AlgorithmFlags

[SWS_CRYPTO_01240] KeyProtection enumeration ⌈The Crypto Stack shall provide an enumeration of key protection flags. The following flags shall be supported:

```
#define CKI_BIT(n) (1 << (n))

enum class KeyProtection : uint8_t
{
  External    = CKI_BIT(1),
  Exportable  = CKI_BIT(2),
  Importable  = CKI_BIT(3),
  Unprotected = CKI_BIT(4)
};
```

The `KeyProtection` values' semantics are described in table 8.3. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02404)*

KeyProtection	Explanation
`External`	The key may be stored externally (e.g. outside of the HSM).
`Exportable`	The key may be exported.
`Importable`	The key may be imported.
`Unprotected`	The key may be handled without protection (i.e. plaintext).

Table 8.3: KeyProtection

[SWS_CRYPTO_01241] KeyType enumeration ⌈ The Crypto Stack shall provide an enumeration of key type flags. The following flags shall be supported:

```
enum class KeyType : uint8_t
{
  Symmetric,
  RSA,
  DH,
  Ecc_NISTp256,
  Ecc_NISTp384,
  Ecc_NISTp521,
  Ecc_Ed25519,
  Ecc_X25519,
  Ecc_Ed448,
  Ecc_X448
};
```

The `KeyType` values' semantics are described in table 8.4. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

KeyType	Explanation
`Symmetric`	The key is usable for symmetric algorithms.
`RSA`	The key is usable for RSA operations.
`DH`	The key is usable for key exchange.
`Ecc_NISTp256`	The key is usable for elliptic curve cryptography.
`Ecc_NISTp384`	The key is usable for elliptic curve cryptography.
`Ecc_NISTp521`	The key is usable for elliptic curve cryptography.
`Ecc_Ed25519`	The key is usable for elliptic curve cryptography.
`Ecc_X25519`	The key is usable for elliptic curve cryptography.
`Ecc_Ed448`	The key is usable for elliptic curve cryptography.
`Ecc_X448`	The key is usable for elliptic curve cryptography.

Table 8.4: KeyType

8.1.2.1.9 KeyManagement interface

[SWS_CRYPTO_01242] KeyManagement interface ⌈ The Crypto Stack shall provide an interface for `KeyManagement` located in the header file defined in [SWS_CRYPTO_01109] in the namespace `ara::crypto`.

```
class KeyManagement
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02101, RS_CRYPTO_02102, RS_CRYPTO_02105)*

[SWS_CRYPTO_01243] GetKey method ⌈ The Crypto Stack shall provide a method to query a `Key` interface by its identifier.

```
virtual Key const& GetKey(uint32_t id) = 0;
```

⌋*(RS_CRYPTO_02403, RS_CRYPTO_02001, RS_CRYPTO_02002)*

[SWS_CRYPTO_01244] Generate method ⌈ The Crypto Stack shall provide a method to generate data into a key. The key to generate data into is identified by the parameter `targetKey`. Optionally a random number generator to be used can be specified in the parameter `random`.

```
virtual void Generate(Key const& targetKey, Random* random = nullptr) =
     0;
```

⌋*(RS_CRYPTO_02101)*

[SWS_CRYPTO_01245] Verify method ⌈ The Crypto Stack shall provide a method to verify the validity of raw key data for a specific key. The key to verify the data for is identified by the parameter `key`. The data to be used for verification is supplied in the parameter `data`.

```
virtual bool Verify(Key const& key, ara::crypto::Span const data) const
     = 0;
```

⌋*(RS_CRYPTO_02102, RS_CRYPTO_02404)*

[SWS_CRYPTO_01246] Import method ⌈ The Crypto Stack shall provide a method to import a key. The key to import the data for is identified by the parameter `targetKey`. The key to use for decrypting the protected key data is identified by the parameter `provisioningKey`. The protected data is supplied in the parameter `data`.

```
virtual void Import(Key const& targetKey, Key const& provisioningKey,
    ara::crypto::Span const data);
```

⌋*(RS_CRYPTO_02102, RS_CRYPTO_02404, RS_CRYPTO_02105)*

[SWS_CRYPTO_01247] Export method ⌈ The Crypto Stack shall provide a method to export a key. The key to export the data from is identified by the parameter `sourceKey`. The key to use for ecnrypting the protected key data is identified by the parameter `migrationKey`. The protected data is supplied in the output parameter `data`.

```
virtual void Export(Key const& sourceKey, Key const& migrationKey, ara
    ::crypto::Span data);
```

⌋*(RS_CRYPTO_02105)*

8.1.2.1.10 KeyDeriviation interface

[SWS_CRYPTO_01248] KeyDeriviation interface ⌈ The Crypto Stack shall provide an interface for `KeyDeriviation` located in the header file defined in [SWS_CRYPTO_01110] in the namespace `ara::crypto`.

```
class KeyDeriviation
```

⌋*(RS_CRYPTO_02103)*

[SWS_CRYPTO_01249] Derive method ⌈ The Crypto Stack shall provide a method to derive key material from a base key. The base key is identified by the parameter `baseKey`. The target key is identified by the parameter `targetKey`. The label to be used for derivation is supplied in the parameter `label`. An optional context of arbitrary data may be supplied in the parameter `context`.

```
virtual void Derive(Key const& baseKey, Key const& targetKey, ara::
    crypto::Span label, ara::crypto::Span* context = nullptr) = 0;
```

⌋*(RS_CRYPTO_02103)*

8.1.2.1.11 KeyExchange interface

[SWS_CRYPTO_01250] KeyExchange interface ⌈ The Crypto Stack shall provide an interface for `KeyExchange` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01111] in the namespace `ara::crypto`.

```
class KeyExchange : KeyedPrimitive
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01251] GetPublicValue method ⌈ The Crypto Stack shall provide a method to get a public value that must be sent to the other party for exchanging keys. The public value is provided in the parameter `pubValue`.

```
virtual void GetPublicValue(ara::crypto::Span<uint8_t> pubValue) = 0;
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01252] Exchange method ⌈ The Crypto Stack shall provide a method to execute the key exchange operation. The public value of the client is provided in the parameter `ourPubVal`. The public value of the other party is provided in the parameter `theirPubVal`. The computed shared key data will be supplied in the key identified by the parameter `sharedKey`.

```
virtual void Exchange(ara::crypto::Span const ourPubVal, ara::crypto::
    Span const theirPubVal, Key const& sharedKey) = 0;
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01312] Primitive Factory ⌈ The Crypto Stack shall provide a factory class for each primitive class to create the actual crypto primitives.

```
class HashFactory {
public:
  //this could also be defined with vendor specific deleter
  using HashPtr = std::unique_ptr<Hash>;

  HashPtr CreateHash(const std::string& hashId);
}
```

⌋*()*

8.1.2.2 CryptoNeed API

The CryptoNeed description is the input for the generation of the *CryptoNeed header files* content. The *CryptoNeed header files* contain classes representing the CryptoNeed and ClientServerInterface referenced in the role requiredInterface. The interface of the class is defined by the ClientServerInterface which in turn is influenced by the associated CryptoNeed.

[SWS_CRYPTO_01301] CryptoNeed class ⌈ The Crypto Stack shall provide the definition of a C++ class named <name>CryptoNeed in the *CryptoNeed header file* defined by [SWS_CRYPTO_01001], where name is the CryptoNeed.shortName.

```
class <CryptoNeed.shortName>CryptoNeed {
...
}
```

⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01302] CryptoNeed class base type ⌈ The CryptoNeed class defined in [SWS_CRYPTO_01201] shall have different base types with regard to the value in CryptoNeed.primitiveFamily.

```
#include "ara/sec/crypto/<name>.h"

class <CryptoNeed.shortName>CryptoNeed : public <baseType> {
...
}
```

The <baseType>s to be used are listed in Table 8.5. ⌋*(RS_CRYPTO_02301)*

primitiveFamily value	<baseType> value	<name> value
ASYMMETRIC_ENCRYPT	Cipher	cipher.h
ASYMMETRIC_DECRYPT	Cipher	cipher.h
SYMMETRIC_ENCRYPT	Cipher	cipher.h
SYMMETRIC_DECRYPT	Cipher	cipher.h
AEAD_ENCRYPT	Cipher	cipher.h
AEAD_DECRYPT	Cipher	cipher.h
SIGNATURE_GENERATE	Signer	signer.h
SIGNATURE_VERIFY	Verifier	verifier.h
MAC_GENERATE	Signer	signer.h
MAC_VERIFY	Verifier	verifier.h
HASH	Hash	hash.h

RANDOM	Random	random.h
KEY_DERIVE	KeyDerivation	key_derivation.h
KEY_EXCHANGE	KeyExchange	key_exchange.h
KEY_MANAGEMENT	KeyManagement	key_management.h

Table 8.5: `CryptoNeed.primitiveFamily` supported for interfaces

The concrete `CryptoNeed` class may be created by the Adaptive Platform's "`ARA::CRYPTO`" implementation (e.g. a factory or factory method). This implementation is responsible for creating the correct binding for a software or hardware isolation mechanism (see section 7.2).

8.2 Modeled API Client Server Interfaces

This chapter lists the `ClientServerInterface`s that are used to access the cryptographic implementation from an Adaptive Application.

[SWS_CRYPTO_01303] Port and ClientServerInterface for CryptoNeeds typed as a Cipher ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Cipher` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	Cipher_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to encipher or decipher data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.6: Port - Cipher_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start	
Description	Starts a streaming context of the cipher.	
Parameter	parameters	
	Description	The parameters for initializing the cipher's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.7: ClientServerInterface Cipher - Method: Start

Name	Update
Description	Updates the streaming context of the cipher and hence transforms a chunk of data.

Parameter	input	
	Description	The input data to be transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.8: ClientServerInterface Cipher - Method: Update

Name	Finish	
Description	Finishes a streaming context of the cipher and retrieves the remaining data.	
Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.9: ClientServerInterface Cipher - Method: Finish

Name	Process	
Description	Single call interface to transform (encipher, decipher) data.	
Parameter	parameters	
	Description	The parameters for initializing the cipher's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	input	
	Description	The input data to be transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.10: ClientServerInterface Cipher - Method: Process

[SWS_CRYPTO_01304] Port and ClientServerInterface for CryptoNeeds typed as a Signer ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type Signer its RPortPrototype shall reference a ClientServerInterface designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	Signer_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	**Interface**	ClientServerInterface
Description	Requires operations to sign data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.11: Port - Signer_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	GetTagSize	
Description	Gets the size of the tag produced by the signer.	
Parameter	tagSize	
	Description	The size of the produced tag in bytes.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.12: ClientServerInterface Signer - Method: GetTagSize

Name	Start	
Description	Starts a streaming context of the signer.	
Parameter	parameters	
	Description	The parameters for initializing the signer's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.13: ClientServerInterface Signer - Method: Start

Name	Update	
Description	Updates the streaming context of the signer and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be processed by the signer.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.14: ClientServerInterface Signer - Method: Update

Name	Finish
Description	Finishes a streaming context of the signer and retrieves the digital signature/authentication tag.

Parameter	output	
	Description	The digital signature/authentication tag.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.15: ClientServerInterface Signer - Method: Finish

Name	Process	
Description	Single call interface to sign data.	
Parameter	parameters	
	Description	The parameters for initializing the signers's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	input	
	Description	The input data to be processed by the signer.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	output	
	Description	The digital signature/authentication tag.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.16: ClientServerInterface Signer - Method: Process

[SWS_CRYPTO_01305] Port and ClientServerInterface for CryptoNeeds typed as a Verifier ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Verifier` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	Verifier_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to verify data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.17: Port - Verifier_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start
Description	Starts a streaming context of the verifier.

Parameter	parameters	
	Description	The parameters for initializing the verifier's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.18: ClientServerInterface Verifier - Method: Start

Name	Update	
Description	Updates the streaming context of the verifier and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be processed by the verifier.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.19: ClientServerInterface Verifier - Method: Update

Name	Finish	
Description	Finishes a streaming context of the verifier and performs the verification of the digital signature/authentication tag.	
Parameter	authenticator	
	Description	The digital signature/authentication tag to verify against.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	length	
	Description	The length in bits of the digital signature/authentication tag to verify the leftmost against.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	result	
	Description	The result of the comparison.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.20: ClientServerInterface Verifier - Method: Finish

Name	Process
Description	Single call interface to verify data.

Parameter	parameters	
	Description	The parameters for initializing the verifier's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	authenticator	
	Description	The digital signature/authentication tag to verify against.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	length	
	Description	The length in bits of the digital signature/authentication tag to verify the leftmost against.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	result	
	Description	The result of the comparison.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.21: ClientServerInterface Verifier - Method: Process

[SWS_CRYPTO_01306] Port and ClientServerInterface for CryptoNeeds typed as a Random ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Random` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	Random_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to generate random data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.22: Port - Random_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Seed	
Description	Provide additional seed.	
Parameter	seed	
	Description	The buffer to hold seed data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.23: ClientServerInterface Random - Method: Seed

Name	Generate

Description	Generate random data.		
Parameter	output		
	Description	The generate random data.	
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.	
	Variation	-	
	Direction	out	

Table 8.24: ClientServerInterface Random - Method: Generate

[SWS_CRYPTO_01307] Port and ClientServerInterface for CryptoNeeds typed as a Hash ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Hash` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	Hash_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to generate hashes.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.25: Port - Hash_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start
Description	Starts a streaming context of the hash.

Table 8.26: ClientServerInterface Hash - Method: Start

Name	Update	
Description	Updates the streaming context of the hash and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be transformed by the hash.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.27: ClientServerInterface Hash - Method: Update

Name	Finish	
Description	Finishes a streaming context of the hash and retrieves the digest.	
Parameter	digest	
	Description	The digest of the data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.28: ClientServerInterface Hash - Method: Finish

Name	Process

Description	Single call operation for hashing.	
Parameter	input	
	Description	The input data to be transformed by the hash.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	digest	
	Description	The digest of the data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.29: ClientServerInterface Hash - Method: Process

[SWS_CRYPTO_01308] Port and ClientServerInterface for CryptoNeeds typed as a KeyDerivation ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyDerivation` its `RPortPrototype` shall reference a `ClientServer-Interface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	KeyDerivation_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to derive a key.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.30: Port - KeyDerivation_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Derive	
Description	Perform a key derivation.	
Parameter	baseKey	
	Description	The key to derive from.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	targetKey	
	Description	The key to be derived.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	label	
	Description	The label to use for the derivation.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Parameter	context	
	Description	The context for the derivation.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.31: ClientServerInterface KeyDerivation - Method: Derive

[SWS_CRYPTO_01309] Port and ClientServerInterface for CryptoNeeds typed as a KeyExchange ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyExchange` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	KeyExchange_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to exchange a shared key.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.32: Port - KeyExchange_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	GetPublicValue	
Description	Retrieve a public value to be sent to the other party.	
Parameter	pubValue	
	Description	Our public value to be sent to the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.33: ClientServerInterface KeyExchange - Method: GetPublicValue

Name	Exchange	
Description	Perform a key exchange.	
Parameter	ourPubVal	
	Description	Our public value sent to the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	theirPubVal	
	Description	The public value received from by the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Parameter	sharedKey	
	Description	The key handle into which to store the computed shared secret key.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.34: ClientServerInterface KeyExchange - Method: Exchange

[SWS_CRYPTO_01310] Port and ClientServerInterface for CryptoNeeds typed as a KeyManagement ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyManagement` its `RPortPrototype` shall reference a `ClientServerInterface` designed as the tables below. ⌋*(RS_CRYPTO_02401)*

Name	KeyManagement		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to manage key.		

Table 8.35: Port - KeyManagement

Name	GetKey	
Description	Retrieve a key handle for the given identification.	
Parameter	keyId	
	Description	Key identification which is unique within the stack.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	key	
	Description	The key handle.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.36: ClientServerInterface KeyManagement - Method: GetKey

Name	Generate	
Description	Generate key data.	
Parameter	targetKey	
	Description	The key handle to generate the key into.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	random	
	Description	The random number generator to use for generating the key data.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.37: ClientServerInterface KeyManagement - Method: Generate

Name	Verify	
Description	Verify the given key data.	
Parameter	key	
	Description	The key handle to check the data against.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The data to check.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	result	
	Description	The result of the check.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.38: ClientServerInterface KeyManagement - Method: Verify

Name	Import	
Description	Import some protected key data into the key store.	
Parameter	targetKey	
	Description	The key handle to import the data into.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	provisioningKey	
	Description	The key handle to use when decrypting the data to import.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The data to import.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.39: ClientServerInterface KeyManagement - Method: Import

Name	Export
Description	Export key data from the key handle.

Parameter	sourceKey	
	Description	The key handle to export data from.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	migrationKey	
	Description	The key handle to use when protecting the data to export.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The exported key data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.40: ClientServerInterface KeyManagement - Method: Export

[SWS_CRYPTO_01311] ClientServerInterface for Keys ⌈ The interfaces described in 8.1.2.1 which are derived from the `KeyedPrimitive` ([SWS_CRYPTO_01201]) and the interface `KeyManagement` ([SWS_CRYPTO_01242]) can return a `Key` handle. This handle cannot be modeled as a `RPortPrototype` to be used by the Adaptive Application. Still operations invoked on the returned `Key` shall behave as if they were modeled referencing a `ClientServerInterface` designed as the tables below. ⌋ *(RS_CRYPTO_02401)*

Name	GetId	
Description	Get the identifier associated with this key.	
Parameter	keyId	
	Description	Key identification which is unique within the stack.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.41: ClientServerInterface Key - Method: GetId

Name	GetType	
Description	Return the type of the key.	
Parameter	type	
	Description	The key htype.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.42: ClientServerInterface Key - Method: GetType

Name	GetSize

Description	Return the key size in bits.	
Parameter	size	
	Description	The key size in bits.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.43: ClientServerInterface Key - Method: GetSize

Name	GetUsage	
Description	Return information on what this key may be used for.	
Parameter	usage	
	Description	The key's usage information.
	Type	uint32
	Variation	-
	Direction	out

Table 8.44: ClientServerInterface Key - Method: GetUsage

Name	GetProtection	
Description	Return information on what the restrictions for handling the key are.	
Parameter	protection	
	Description	The key protection flags.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.45: ClientServerInterface Key - Method: GetProtection

9 Direct API specification

The Direct Crypto API supports the streaming interface approach that supported by methods with following names:

- The `start()` methods reset the internal states of the cryptographic context to begin processing chunks of data.
- The `update()` methods update the internal state of the cryptographic context by processing the given chunk of data, but doesn't modify the data.
- The `processBlocks()` and `processBytes()` methods process the given chunk of data and may update internal state of the context (in most cases they encrypt or decrypt the data chunk and additionally may authenticate it).
- The `finish()` methods conclude the operation cycle of the algorithm and may return a result of the transformation accumulated in the context (they are used for finalization of hashing or authentication).
- The `finishBytes()` methods conclude the data stream processing and as a result it may return on the output a data chunk of larger size than was provided on the input (they are used for finalization of encryption/decryption).

Direct Crypto API header files are located in the directory "`ara/crypto/direct/`". Each functional sub-domain of the Direct Crypto API is represented by own namespace. Files of each namespace are located in a sub-directory with correspondent name. Each of these namespace-dedicated sub-directories contains a header file named "`EntryPoint.h`". It is enough to include only this file for usage of the whole functionality of correspondent sub-domain of the Direct API. For example following set of the include directives grants access to whole functionality of the Direct Crypto API:

```
#include "ara/crypto/direct/crypt/EntryPoint.h"
#include "ara/crypto/direct/keys/EntryPoint.h"
#include "ara/crypto/direct/x509/EntryPoint.h"
```

9.1 Cryptographic Primitives Naming Convention

Future Crypto Providers can support some crypto algorithms that are not well-known/standardized today, therefore this API specification doesn't provide a concrete list of crypto algorithms' identifiers and doesn't suppose usage of numerical identifiers. Instead of this a provider supplier should provide string names of supported algorithms in accompanying documentation. I.e. a concrete set of crypto algorithms supported by Crypto Provider remain at the discretion of a supplier.

Any name of a crypto algorithm should satisfy to following rules

1. Only latin alphanumeric characters and 6 defined below delimiters can be used for crypto algorithm definition.

2. Case of letters does not matter, i.e. all comparisons of the identifiers must be always case-insensitive.

3. Any name of a crypto algorithm should satisfy to following structure:

   “{TargetTransformation(Mode)} / {SupportingAlgorithms} / {Encoding&Padding}”

   where

   - “{TargetTransformation(Mode)}” – a specificator of target transformation: for complex transformations it is a mode name, but for fully-defined algorithms it is just their name;
   - “{SupportingAlgorithms}” – a specificator of basic crypto algorithm(s) including key length and/or block length;
   - “{Encoding&Padding}” – a specificator of encoding and/or padding method. It can support following predefined name (equal to empty specification):
   - “Zero” – a default encoding & padding method: if data are already aligned to the block boundary then doesn't add anything, but if they are not aligned then applies a padding by '\0' bytes up to the block boundary.

4. Allowed delimiters:

   - '/' – separator between main components of the whole algorithm specification.
   - '_' – separator instead of general separation characters (e.g.: ' ', '.', ':', '-', '/') in original name of standard; this delimiter can be applied between two digits or two letters only!
   - '-' – separator between a base algorithm name and its precise specificators that define key-length or block-length in bits.
   - '+' – separator between a few base algorithms' specifications for a cascade transformation definition.
   - ',' – separator between a few base algorithms' specifications for a case if the whole algorithm is based on a few types of basic transformations.
   - '.' – separator between a common name of a standard and its specific part or its version that precises a specification of concrete transformation.

   Examples of well-known algorithm names: “ECDSA-256”, “ECDH-256”, “AES-128”, “Camellia-256”, “3DES-168”, “ChaCha20”, “GOST28147_89”, “SHA1”, “SHA2-256”, “GOSTR3410.94”, “GOSTR3410.2001”, “GOSTR3410.2012-512”.

   Examples of well-known modes names: “ECB”, “OFB”, “CFB”, “CBC”, “PCBC”, “CTR”, “HMAC”, “CBC_MAC”, “OMAC1”, “OMAC2”, “VMAC”, `"Poly1305"`, `"CCM"`, `"GCM"`, `"OCB"`, `"CWC"`, `"EAX"`, `"KDF1"`, `"KDF2"`, `"KDF3"`, `"MGF1"`.

Examples of the encoding and padding names: `"ANSI_X923"`, `"ISO10126"`, `"PKCS7"`, `"ISO_IEC7816_4"`, `"PKCS1.v1_5"`, `"OAEP"`, `"OAEPplus"`, `"SAEP"`, `"SAEPplus"`, `"PSS"`, `"EME"`, `"EMSA"`.

Examples of fully defined transformations:

- `"ECDSA-384"` means ECDSA signature algorithm with private key-length 384 bit.
- `"ECDH-512"` means ECDH key agreement algorithm with private key-length 512 bit.
- `"CTR/AES-256"` means a CTR-mode stream cipher based on AES algorithm with key-length 256 bit.
- `"CBC/AES-192+Camellia-192/PKCS7"` means CBC-mode cipher based on cascade application of AES-192 and Camellia-192 with padding of last block according to PKCS#7.
- `"HMAC/SHA-256"` means HMAC based on SHA-256.

5. If an algorithm support a few variable length parameters then they should be specified in following order: key, I/O-block or output digest, IV or input block (e.g.: `"Kalyna-512-256"` means block cipher Kalina with 512-bit key and 256-bit block).
6. If a transformation is based on a few basic cryptoalgorithms then they should be specified in an order corresponding to the level of their application (see example below for RSA).
7. Following Mode specificators can be used for RSA-based algorithms:
   - `"SIG"` – signature primitive (e.g.: `"SIG/RSA-2048,SHA-160/PKCS1.v1_5,EMSA"`)
   - `"VER"` – verification primitive (e.g.: `"VER/RSA-2048,SHA-160/PKCS1.v1_5,EMSA"`)
   - `"ENC"` – encryption primitive (e.g.: `"ENC/RSA-2048,MGF1,SHA-160/PKCS1.v1_5,EME"`, `"ENC/RSA-4096,MGF1,SHA2-256/OAEP,EME"`)
   - `"DEC"` – decryption primitive (e.g.: `"DEC/RSA-2048,MGF1,SHA-160/PKCS1.v1_5,EME"`, `"DEC/RSA-4096,MGF1,SHA2-256/OAEP,EME"`)
   - `"KEM"` – Key Encapsulation Mechanism (e.g.: `"KEM/RSA-2048,AES-128,KDF3,SHA-256"`)

8. A supplier should strive to use shortest names of algorithms, sufficient for their unambiguous identification.

9.2 ara Namespace Reference

Namespaces

- crypto

9.2.1 Detailed Description

AUTOSAR Adaptive Platform namespace.

9.3 ara::crypto Namespace Reference

Namespaces

- crypt
- keys
- x509

Classes

- struct COUID
- struct CustomDeleter
- interface CustomDisposable
- interface ProviderInfo
- interface SecAccessViolation
- interface SecErrorInfo
- interface SecInvalidArgument
- interface SecLogicError
- interface SecRuntimeError
- interface TrustedContainer
- struct UUID

Typedefs

- using ActorUID_t = UUID
- using CryptoProviderUID_t = UUID
- using LogicalSlotUID_t = UUID
- using CryptoAlgId_t = std::uint64_t
- using AllowedUsageFlags_t = std::uint32_t
- using GUID_t = UUID
- typedef std::uint8_t byte_t

Enumerations

Functions

- bool operator== (const COUID &lhs, const COUID &rhs)
- bool operator< (const COUID &lhs, const COUID &rhs)
- bool operator> (const COUID &lhs, const COUID &rhs)
- bool operator!= (const COUID &lhs, const COUID &rhs)
- bool operator<= (const COUID &lhs, const COUID &rhs)
- bool operator>= (const COUID &lhs, const COUID &rhs)
- bool operator== (const UUID &lhs, const UUID &rhs)
- bool operator< (const UUID &lhs, const UUID &rhs)
- bool operator> (const UUID &lhs, const UUID &rhs)
- bool operator!= (const UUID &lhs, const UUID &rhs)
- bool operator<= (const UUID &lhs, const UUID &rhs)
- bool operator>= (const UUID &lhs, const UUID &rhs)

Variables

- const AllowedUsageFlags_t ALLOW_KDF_MATERIAL_ANY_USAGE

9.3.1 Detailed Description

This namespace contains whole Cryptography related API splitted on functional sub-domains (daughter namespaces).

9.3.2 Typedef Documentation

9.3.2.1 using ActorUID_t = UUID

[SWS_CRYPTO_10001] Interface definition: ⌈ Unique ID of the Actor process (it is a persistent UID defined on the application design phase).

Note

Actor UID can be associated with "User" or "Owner" process permissions.

⌋*(RS_CRYPTO_02110, RS_CRYPTO_02114, RS_CRYPTO_02404)*

9.3.2.2 using CryptoProviderUID_t = UUID

[SWS_CRYPTO_10002] Interface definition: ⌈ Unique ID of a Crypto Provider (it is a persistent UID defined on the provider design phase).

⌋*(RS_CRYPTO_02401, RS_CRYPTO_02404)*

9.3.2.3 using LogicalSlotUID_t = UUID

[SWS_CRYPTO_10003] Interface definition: ⌈ Logical Slot UID (it is a persistent UID defined on the application design phase).

⌋*(RS_CRYPTO_02404, RS_CRYPTO_02405)*

9.3.2.4 using CryptoAlgId_t = std::uint64_t

[SWS_CRYPTO_10004] Interface definition: ⌈ Crypto Algorithm Identifier type.

⌋*(RS_CRYPTO_02102, RS_CRYPTO_02107, RS_CRYPTO_02404)*

9.3.2.5 using AllowedUsageFlags_t = std::uint32_t

[SWS_CRYPTO_10005] Interface definition: ⌈ A container type for bit-flags of allowed usages of a key or a secret seed object.

Note

> Only directly specified usages of a key are allowed, all other are prohibited! Similar set of flags are defined for the usage restrictions of original key/seed and for a symmetric key or seed that potentially can be derived from the original one. A symmetric key or secret seed can be derived from the original one, only if it supports ALLOW_KEY_AGREEMENT or ALLOW_KEY_DIVERSIFY or ALLOW_KEY_DERIVATION!

⌋*(RS_CRYPTO_02111, RS_CRYPTO_02404)*

9.3.2.6 using GUID_t = UUID

The Globally Unique Identifier (GUID) is an alias of Universally Unique Identifier (UUID).

9.3.2.7 typedef std::uint8_t byte_t

A unified type definition for a single byte.

9.3.2.8 typedef std::vector<byte_t> ReadWriteMemRegion

Read-Write Memory Region (intended for [in/out] arguments)

9.3.2.9 typedef std::vector<byte_t> WritableMemRegion

Writable Memory Region (intended for [out] arguments)

9.3.2.10 typedef const std::vector<byte_t> ReadOnlyMemRegion

Read-Only Memory Region (intended for [in] arguments)

9.3.3 Enumeration Type Documentation

9.3.3.1 enum CryptoObjectType : std::uint8_t `[strong]`

[SWS_CRYPTO_10006] Interface definition: ⌈ Enumeration of all types of crypto objects, i.e. types of content that can be stored to a key slot.

⌋*(RS_CRYPTO_02004)*

Enumerator

NONE	Used for empty containers (key slots) and in a case of the dependency absence.
UNKNOWN	Object type unknown (meaning is identical to the previous one)
DOMAIN_PARAMETERS	Domain Parameters object.
SYMMETRIC_KEY	Symmetric Key object.
PRIVATE_KEY	Private Key object.
PUBLIC_KEY	Public Key object.
SIGNATURE	Digital Signature (or MAC/HMAC) object.
PASSWORD_HASH	Password Hash object (hash diversified by a random seed)
SECRET_SEED	Secret Seed object (Note: the Seed cannot have an associated crypto algorithm!)
CERT_REQUEST	Certification Request object.
CERTIFICATE	Certificate object (Not supported yet!)

9.3.3.2 enum ErrorDomain : std::uint32_t `[strong]`

[SWS_CRYPTO_19000] Interface definition: ⌈ Enumeration of all known error domains.

⌋*(RS_CRYPTO_02310)*

Enumerator

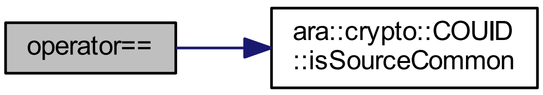
UNDEFINED	Undefined/Unknown errors domain.
CRYPTO_PROVIDER	Crypto Provider errors domain.
KEY_STORAGE	Key Storage Provider errors domain.
X509_PROVIDER	X.509 Provider errors domain.

9.3.4 Function Documentation

9.3.4.1 bool ara::crypto::operator== (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

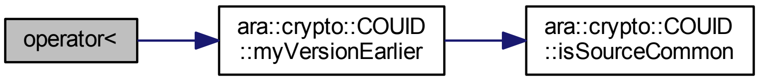
Here is the call graph for this function:



9.3.4.2 bool ara::crypto::operator< (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

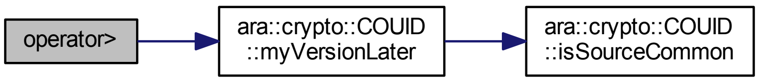
Here is the call graph for this function:


operator<
ara::crypto::COUID ::myVersionEarlier
ara::crypto::COUID ::isSourceCommon

9.3.4.3 bool ara::crypto::operator> (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

Here is the call graph for this function:


operator>
ara::crypto::COUID ::myVersionLater
ara::crypto::COUID ::isSourceCommon

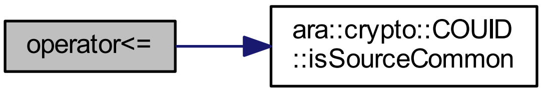
9.3.4.4 bool ara::crypto::operator!= (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.5 bool ara::crypto::operator<= (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

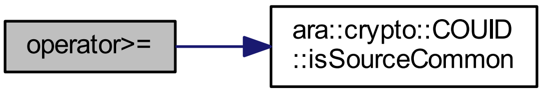
Here is the call graph for this function:



9.3.4.6 bool ara::crypto::operator>= (const COUID & *lhs*, const COUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

Here is the call graph for this function:



9.3.4.7 bool ara::crypto::operator== (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.8 bool ara::crypto::operator< (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.9 bool ara::crypto::operator> (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.10 bool ara::crypto::operator!= (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.11 bool ara::crypto::operator<= (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.4.12 bool ara::crypto::operator>= (const UUID & *lhs*, const UUID & *rhs*) `[inline]`

Standard comparison operator with default behavior.

9.3.5 Variable Documentation

9.3.5.1 const CryptoAlgId_t ALGID_UNDEFINED = 0ULL

Algorithm ID is undefined.

9.3.5.2 const CryptoAlgId_t ALGID_ANY = ALGID_UNDEFINED

Any Algorithm ID is allowed.

9.3.5.3 const CryptoAlgId_t ALGID_DEFAULT = ALGID_UNDEFINED

Default Algorithm ID (in current context/primitive).

9.3.5.4 const CryptoAlgId_t ALGID_NONE = ALGID_UNDEFINED

None of Algorithm ID (i.e. an algorithm definition is not applicable).

9.3.5.5 const AllowedUsageFlags_t ALLOW_PROTOTYPED_ONLY = 0L

The key/seed usage will be fully specified by a key slot prototype (the object can be used only after reloading from the slot).

9.3.5.6 const AllowedUsageFlags_t ALLOW_DATA_ENCRYPTION = 0x0001L

The key/seed can be used for data encryption initialization (applicable to symmetric and asymmetric keys).

9.3.5.7 const AllowedUsageFlags_t ALLOW_DATA_DECRYPTION = 0x0002L

The key/seed can be used for data decryption initialization (applicable to symmetric and asymmetric keys).

9.3.5.8 const AllowedUsageFlags_t ALLOW_SIGNATURE = 0x0004L

The key/seed can be used for digital signature or MAC/HMAC production (applicable to symmetric and asymmetric keys).

9.3.5.9 const AllowedUsageFlags_t ALLOW_VERIFICATION = 0x0008L

The key/seed can be used for digital signature or MAC/HMAC verification (applicable to symmetric and asymmetric keys).

9.3.5.10 const AllowedUsageFlags_t ALLOW_KEY_AGREEMENT = 0x0010L

The seed or asymmetric key can be used for key-agreement protocol execution.

9.3.5.11 const AllowedUsageFlags_t ALLOW_KEY_DIVERSIFY = 0x0020L

The seed or symmetric key can be used for slave-keys diversification.

9.3.5.12 const AllowedUsageFlags_t ALLOW_DRNG_INIT = 0x0040L

The seed or symmetric key can be used for initialization of a Deterministic Random Number Generators (DRNG).

9.3.5.13 const AllowedUsageFlags_t ALLOW_KDF_MATERIAL = 0x0080L

The seed or symmetric key can be used as a `KeyMaterial` for slave-keys derivation via a Key Derivation Function (KDF).

9.3.5.14 const AllowedUsageFlags_t ALLOW_KEY_EXPORTING = 0x0100L

The key can be used as "transport" one for Key-Wrap or Encapsulate transformations (applicable to symmetric and asymmetric keys).

9.3.5.15 const AllowedUsageFlags_t ALLOW_KEY_IMPORTING = 0x0200L

The key can be used as "transport" one for Key-Unwrap or Decapsulate transformations (applicable to symmetric and asymmetric keys).

9.3.5.16 const AllowedUsageFlags_t ALLOW_EXACT_MODE_ONLY = 0x8000L

The key can be used only for the mode directly specified by `Key::AlgId_t`.

9.3.5.17 const AllowedUsageFlags_t ALLOW_DERIVED_DATA_ENCRYPTION = ALLOW_DATA_ENCRYPTION << 16

A derived seed or symmetric key can be used for data encryption.

9.3.5.18 const AllowedUsageFlags_t ALLOW_DERIVED_DATA_DECRYPTION = ALLOW_DATA_DECRYPTION << 16

A derived seed or symmetric key can be used for data decryption.

9.3.5.19 const AllowedUsageFlags_t ALLOW_DERIVED_SIGNATURE = ALLOW_SIGNATURE << 16

A derived seed or symmetric key can be used for MAC/HMAC production.

9.3.5.20 const AllowedUsageFlags_t ALLOW_DERIVED_VERIFICATION = ALLOW_VERIFICATION << 16

A derived seed or symmetric key can be used for MAC/HMAC verification.

9.3.5.21 const AllowedUsageFlags_t ALLOW_DERIVED_KEY_DIVERSIFY = ALLOW_KEY_DIVERSIFY << 16

A derived seed or symmetric key can be used for slave-keys diversification.

9.3.5.22 const AllowedUsageFlags_t ALLOW_DERIVED_DRNG_INIT = ALLOW_DRNG_INIT << 16

A derived seed or symmetric key can be used for initialization of a Deterministic Random Number Generators (DRNG).

9.3.5.23 const AllowedUsageFlags_t ALLOW_DERIVED_KDF_MATERIAL = ALLOW_KDF_MATERIAL << 16

A derived seed or symmetric key can be used as a `KeyMaterial` for slave-keys derivation via a Key Derivation Function (KDF).

9.3.5.24 const AllowedUsageFlags_t ALLOW_DERIVED_KEY_EXPORTING = ALLOW_KEY_EXPORTING << 16

A derived seed or symmetric key can be used as a "transport" one for Key-Wrap transformation.

9.3.5.25 const AllowedUsageFlags_t ALLOW_DERIVED_KEY_IMPORTING = ALLOW_KEY_IMPORTING << 16

A derived seed or symmetric key can be used as a "transport" one for Key-Unwrap transformation.

9.3.5.26 const AllowedUsageFlags_t ALLOW_DERIVED_EXACT_MODE_ONLY = ALLOW_EXACT_MODE_ONLY << 16

A derived seed or symmetric key can be used only for the mode directly specified by `Key::AlgId_t`.

9.3.5.27 const AllowedUsageFlags_t ALLOW_KDF_MATERIAL_ANY_USAGE

Initial value:

```
= ALLOW_KDF_MATERIAL
        | ALLOW_DERIVED_DATA_ENCRYPTION |
      ALLOW_DERIVED_DATA_DECRYPTION |
      ALLOW_DERIVED_SIGNATURE | ALLOW_DERIVED_VERIFICATION
        | ALLOW_DERIVED_KEY_DIVERSIFY |
      ALLOW_DERIVED_DRNG_INIT | ALLOW_DERIVED_KDF_MATERIAL |
      ALLOW_DERIVED_KEY_EXPORTING |
      ALLOW_DERIVED_KEY_IMPORTING
```

The seed or symmetric key can be used as a `KeyMaterial` for a Key Derivation Function (KDF) and the derived "slave" keys can be used without limitations.

9.4 ara::crypto::crypt Namespace Reference

Classes

- interface AuthStreamCipherCtx
- interface BlockCryptor
- interface BufferedDigest
- interface CryptoBadAlloc
- interface CryptoContext
- interface CryptoInvalidArgument
- interface CryptoLogicError
- interface CryptoObject
- interface CryptoPrimitiveId
- interface CryptoProvider
- interface CryptoRuntimeError
- interface CryptoUnsupported
- interface CryptoUsageViolation
- interface DecryptPrivateCtx
- interface DomainParameters
- interface EncryptPublicCtx
- interface HashFunctionCtx
- interface Key
- interface KeyAgreePrivateCtx
- interface KeyDecapsulatePrivateCtx
- interface KeyDeriveFuncCtx
- interface KeyDiversifierCtx
- interface KeyedContext
- interface KeyEncapsulatePublicCtx
- interface KeyEncapsulator
- interface KeyMaterial
- interface MessageAuthCodeCtx
- interface MsgRecoveryPublicCtx

- interface PasswordCacheCtx
- interface PasswordHash
- interface PrivateKey
- interface PrivateKeyContext
- interface PublicKey
- interface PublicKeyContext
- interface RandomGeneratorCtx
- interface SecretSeed
- interface Serializable
- interface SigEncodePrivateCtx
- interface Signature
- interface SignatureHandler
- interface SignPrivateCtx
- interface StreamCipherCtx
- interface SymmetricBlockCipherCtx
- interface SymmetricKey
- interface SymmetricKeyContext
- interface SymmetricKeyWrapCtx
- interface VerifyPublicCtx
- interface X509AlgorithmId
- interface X509CertRequest
- interface X509PublicKeyInfo
- interface X509RequestSignerCtx
- interface X509Signature

Typedefs

- using ReservedContextIndex_t = std::size_t
- using ReservedObjectIndex_t = std::size_t

Enumerations

Functions

- CryptoProvider::sptr_t loadCryptoProvider (const CryptoProviderUID_t ∗providerUid=nullptr) noexcept(false)

9.4.1 Detailed Description

This namespace contains Crypto Provider interfaces.

9.4.2 Typedef Documentation

9.4.2.1 using ReservedContextIndex_t = std::size_t

[SWS_CRYPTO_20002] Interface definition: ⌈ Type of Reserved Context Index (maximal value means "NOT RESERVED").

Note

> Values of this type should be used for indexing slots reserved for Cryptographic Contexts in the "Memory Pool".

⌋*(RS_CRYPTO_02311)*

9.4.2.2 using ReservedObjectIndex_t = std::size_t

[SWS_CRYPTO_20003] Interface definition: ⌈ Type of Reserved Object Index (maximal value means "NOT RESERVED").

Note

> Values of this type should be used for indexing slots reserved for Cryptographic Objects in the "Memory Pool".

⌋*(RS_CRYPTO_02311)*

9.4.3 Enumeration Type Documentation

9.4.3.1 enum KeyType : std::uint32_t `[strong]`

[SWS_CRYPTO_20001] Interface definition: ⌈ Enumeration of all known types of supported keys.

⌋*(RS_CRYPTO_02311)*

Enumerator

UNKNOWN	A value reserved for erroneous situations.
SYMMETRIC_KEY	SymmetricKey.
PRIVATE_KEY	PrivateKey.
PUBLIC_KEY	PublicKey.

9.4.4 Function Documentation

9.4.4.1 CryptoProvider::sptr_t ara::crypto::crypt::loadCryptoProvider (const CryptoProviderUID_t ∗ *providerUid* = `nullptr`) `[noexcept]`

[SWS_CRYPTO_20000] Interface definition: ⌈ Factory that creates or return existing single instance of specific Crypto Provider.

Parameters

`in`	*providerUid*	A globally unique identifier of required Crypto Provider.

Returns

Shared smart pointer to loaded Crypto Provider.

Note

If `(providerUid == nullptr)` then platform default provider should be loaded.

Exceptions

CryptoRealtimeError	If requested Crypto Provider cannot be loaded.

⌋*(RS_CRYPTO_02401)*

9.4.5 Variable Documentation

9.4.5.1 const ReservedContextIndex_t ALLOC_CONTEXT_ON_HEAP = static_cast<ReservedContextIndex_t>(-1) `[static]`

Marker that the Context should be allocated on the heap.

9.4.5.2 const ReservedObjectIndex_t ALLOC_OBJECT_ON_HEAP = static_cast<ReservedObjectIndex_t>(-1) `[static]`

Marker that the Object should be allocated on the heap.

9.5 ara::crypto::keys Namespace Reference

Classes

- interface KeysAccessViolation

- struct KeySlotContentProps
- struct KeySlotPrototypeProps
- interface KeyStorageProvider
- struct UserPermissions

Enumerations

Functions

- KeyStorageProvider::sptr_t loadKeyStorageProvider () noexcept(false)

9.5.1 Detailed Description

This namespace contains Key Storage Provider interfaces

9.5.2 Enumeration Type Documentation

9.5.2.1 enum VersionControlType : std::uint8_t `[strong]`

[SWS_CRYPTO_30001] Interface definition: ⌈ Enumeration of all Version Control Types.

⌋*(RS_CRYPTO_02116)*

Enumerator

NONE Version control is not applied for content of this slot.
LOCAL Version control is applied, slot initialization is not required, but only locally produced crypto object can be saved to the slot.
EXTERNAL Version control is applied, slot must be initialized by a COUID, specifying concrete external source of objects and minimal version.
SWITCH_TO_LOCAL Similar to LOCAL, but the slot can be initialized by an externally produced crypto object, which doesn't leave version control "track".

9.5.3 Function Documentation

9.5.3.1 KeyStorageProvider::sptr_t ara::crypto::keys::loadKeyStorageProvider ( ) `[noexcept]`

[SWS_CRYPTO_30000] Interface definition: ⌈ Factory that creates or return existing single instance of the Key Storage Provider.

Returns

Shared smart pointer to loaded Key Storage Provider.

Exceptions

CryptoRealtimeError	If a Crypto Provider instance cannot be created.

⌋*(RS_CRYPTO_02109, RS_CRYPTO_02401)*

9.6 ara::crypto::x509 Namespace Reference

Classes

- interface BasicCertInfo
- interface Certificate
- interface CertificateRequest
- interface X509BadAlloc
- interface X509DN
- interface X509Extensions
- interface X509InvalidArgument
- interface X509LogicError
- interface X509Provider
- interface X509RuntimeError
- interface X509Unsupported

Functions

- X509Provider::sptr_t loadX509Provider () noexcept(false)

9.6.1 Detailed Description

This namespace contains X.509 Provider interfaces

9.6.2 Function Documentation

9.6.2.1 X509Provider::sptr_t ara::crypto::x509::loadX509Provider ( ) [noexcept]

[SWS_CRYPTO_40000] Interface definition: ⌈ Factory that creates or return existing single instance of the X.509 Provider.

Returns

Shared smart pointer to loaded X.509 Provider.

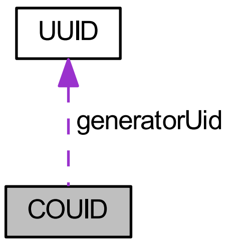
Exceptions

X509RealtimeError	If the X.509 Provider cannot be loaded.

⌋*(RS_CRYPTO_02306)*

9.7 COUID Struct Reference

Collaboration diagram for COUID:



Public Member Functions

- COUID ()
- bool isSourceCommon (const COUID &anotherId) const
- bool myVersionEarlier (const COUID &anotherId) const
- bool myVersionLater (const COUID &anotherId) const
- COUID (const COUID &)=default
- COUID (COUID &&)=delete

- COUID & operator= (const COUID &)=default
- COUID & operator= (COUID &&)=delete

9.7.1 Detailed Description

[SWS_CRYPTO_10100] Interface definition: ⌈ Definition of Crypto Object Unique Identifier (COUID) type.

⌋*(RS_CRYPTO_02005, RS_CRYPTO_02006, RS_CRYPTO_02404)*

9.7.2 Constructor & Destructor Documentation

9.7.2.1 COUID ( ) `[inline]`

[SWS_CRYPTO_10110] Interface definition: ⌈ Default constructor (initializes the identifier by zeros).

⌋*(RS_CRYPTO_02311)*

9.7.2.2 COUID (const COUID &) `[default]`

Default behavior of move and copy.

9.7.2.3 COUID (COUID &&) `[delete]`

Default behavior of move and copy.

9.7.3 Member Function Documentation

9.7.3.1 COUID& operator= (const COUID &) `[default]`

Default behavior of move and copy.

9.7.3.2 COUID& operator= (COUID &&) `[delete]`

Default behavior of move and copy.

9.7.3.3 bool isSourceCommon (const COUID & *anotherId*) const `[inline]`

[SWS_CRYPTO_10111] Interface definition: ⌈ Checks whether this identifier has a common source with the one provided by the argument.

Parameters

`in`	*anotherId*	Another identifier for comparison.

Returns

true if both identifiers has common source (identical value of the generatorUid field).

⌋ *(RS_CRYPTO_02006)*

9.7.3.4 bool myVersionEarlier (const COUID & *anotherId*) const `[inline]`

[SWS_CRYPTO_10112] Interface definition: ⌈ Checks whether this identifier was generated earlier than the one provided by the argument.

Parameters

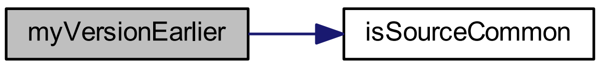
`in`	*anotherId*	Another identifier for comparison.

Returns

true if this identifier was generated earlier than the anotherId.

⌋ *(RS_CRYPTO_02006)*

Here is the call graph for this function:



9.7.3.5 bool myVersionLater (const COUID & *anotherId*) const `[inline]`

[SWS_CRYPTO_10113] Interface definition: ⌈ Checks whether this identifier was generated later than the one provided by the argument.

Parameters

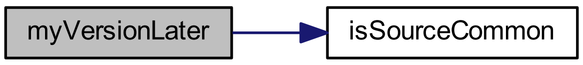
in	*anotherId*	Another identifier for comparison.

Returns

true if this identifier was generated later than the anotherId.

⌋(*RS_CRYPTO_02006*)

Here is the call graph for this function:



9.7.4 Member Data Documentation

9.7.4.1 GUID_t generatorUid

UUID of a generator that has produced this COUID. This UUID can be associated with HSM, physical host/ECU or VM.

9.7.4.2 std::uint64_t versionStamp

Sequential value of a steady timer or simple counter, representing version of correspondent Crypto Object.

9.8 CustomDeleter Struct Reference

Public Member Functions

- constexpr CustomDeleter () noexcept(true)=default
- void operator() (CustomDisposable ∗ptr) const noexcept(true)

9.8.1 Detailed Description

[SWS_CRYPTO_10300] Interface definition: ⌈ A custom deleter definition.

⌋(*RS_CRYPTO_02404*)

9.8.2 Constructor & Destructor Documentation

9.8.2.1 constexpr CustomDeleter ( ) `[default]`, `[noexcept]`

[SWS_CRYPTO_10311] Interface definition: ⌈ Constructor of the Custom Deleter.

⌋(*RS_CRYPTO_02311*)

9.8.3 Member Function Documentation

9.8.3.1 void operator() (CustomDisposable ∗ *ptr*) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_10312] Interface definition: ⌈ Custom Delete operator.

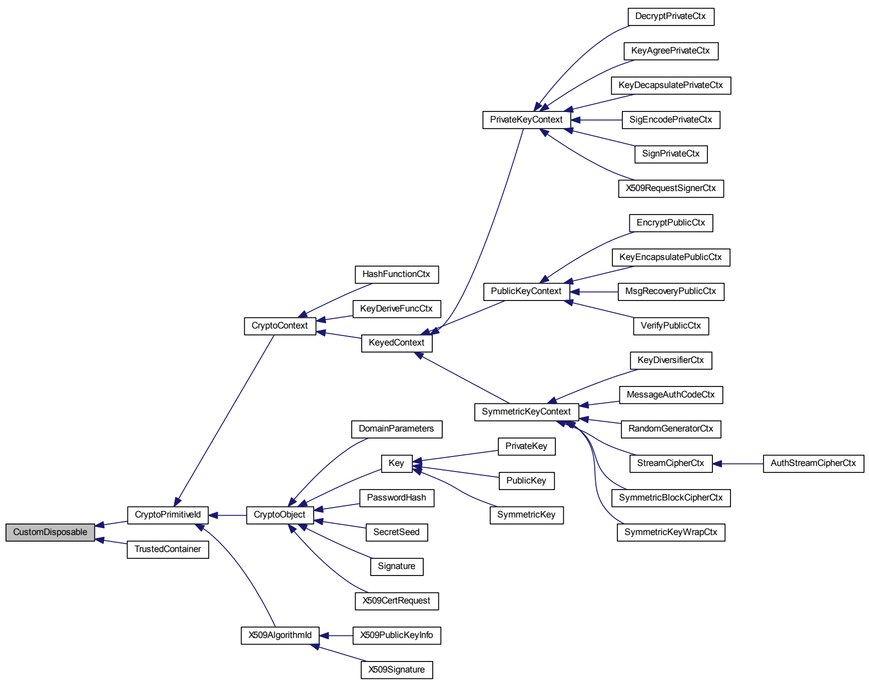
Parameters

`in`	*ptr*	A pointer to an instance of the `CustomDisposable` interface.

⌋(*RS_CRYPTO_02311*)

9.9 CustomDisposable Interface Reference

Inheritance diagram for CustomDisposable:



Public Member Functions

- virtual void release () noexcept(true)=0

Protected Member Functions

- virtual ∼CustomDisposable ()=default

Static Protected Member Functions

- static void operator delete (void ∗p, std::size_t n)
- static void operator delete (void ∗p)
- static void operator delete[ ] (void ∗p, std::size_t n)
- static void operator delete[ ] (void ∗p)

9.9.1 Detailed Description

[SWS_CRYPTO_10200] Interface definition: ⌈ A basic interface of customly disposable objects.

⌋*(RS_CRYPTO_02404)*

9.9.2 Constructor & Destructor Documentation

9.9.2.1 virtual ∼CustomDisposable ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_10210] Interface definition: ⌈ Hides destructor client side code.

⌋*(RS_CRYPTO_02311)*

9.9.3 Member Function Documentation

9.9.3.1 virtual void release ( ) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_10211] Interface definition: ⌈ Release allocated memory and other resources.

⌋*(RS_CRYPTO_02404)*

9.9.3.2 void operator delete (void ∗ *p*, std::size_t *n*) `[inline]`, `[static]`, `[protected]`

Hides delete operator from client side code.

9.9.3.3 void operator delete (void ∗ *p*) `[inline]`, `[static]`, `[protected]`

Hides delete operator from client side code.

9.9.3.4 void operator delete[] (void ∗ *p*, std::size_t *n*) `[inline]`, `[static]`, `[protected]`

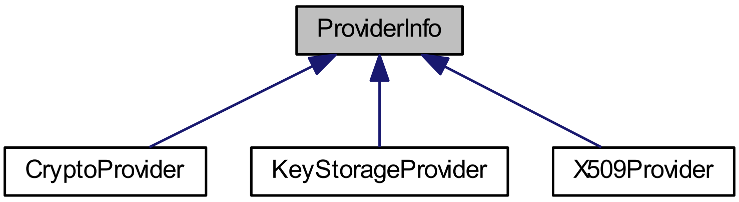
Hides delete operator from client side code.

9.9.3.5 void operator delete[] (void ∗ p) `[inline]`, `[static]`, `[protected]`

Hides delete operator from client side code.

9.10 ProviderInfo Interface Reference

Inheritance diagram for ProviderInfo:



Public Member Functions

- virtual std::uint64_t getProviderVersion () const noexcept(true)=0
- virtual std::size_t getProviderName (std::string ∗name=nullptr) const noexcept(false)=0
- virtual void getProviderId (GUID_t &providerId) const noexcept(true)=0

Protected Member Functions

- virtual ∼ProviderInfo ()=default

9.10.1 Detailed Description

[SWS_CRYPTO_10500] Interface definition: ⌈ A common interface for obtaining an identification information of a Provider.

⌋*(RS_CRYPTO_02311)*

9.10.2 Constructor & Destructor Documentation

9.10.2.1 virtual ∼ProviderInfo ( ) `[protected], [virtual], [default]`

[SWS_CRYPTO_10510] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.10.3 Member Function Documentation

9.10.3.1 virtual std::uint64_t getProviderVersion ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10511] Interface definition: ⌈ Returns an encoded version of the Provider.

Returns

Encoded version that has following structure:

Note

The QWORD value returned by this method includes two DWORD-length fields:

- "Version structure" (32 bits) - most significant DWORD
- "Build date & time" (32 bits) - least significant DWORD (LSDW) The "Version structure" includes four fields (each of them corresponds to one byte):
- Major version (8 bits) - most significant byte (MSB) of the Version DWORD
- Minor version (8 bits)
- Patch version (8 bits)
- Revision version (8 bits) - least significant byte (LSB) of the Version DWORD The "Build date & time" is a number of minutes since 1st of January 1970 00:00. So generally the Version QWORD has following structure:

MSB	8 bits	8 bits	LSB	LSDW (32 bits)
Major	Minor	Patch	Revis	Build date & time

⌋*(RS_CRYPTO_02404)*

9.10.3.2 virtual std::size_t getProviderName (std::string ∗ *name* = `nullptr`) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10512] Interface definition: ⌈ Returns a human readable name of the Provider.

Parameters

`out`	*name*	An optional pointer to a string where Provider Name should be saved.

Returns

Actual length of the Provider Name in chars.

Note

This method doesn't change the capacity of the output buffer! But its size can be changed.

Exceptions

SecInvalidArgument	if `(name != nullptr)`, but the `name->capacity()` is not enough to store the Provider Name.

⌋(*RS_CRYPTO_02311*)

9.10.3.3 virtual void getProviderId (GUID_t & *providerId*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10513] Interface definition: ⌈ Returns the Provider's Globally Unique Identifier (GUID).

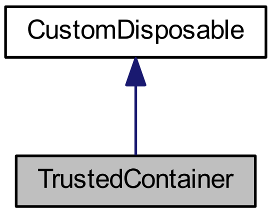
Parameters

`out`	*providerId*	An output buffer for the Provider's GUID.

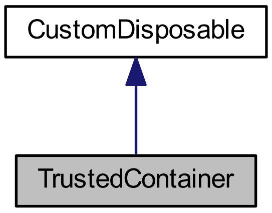
⌋(*RS_CRYPTO_02401*, *RS_CRYPTO_02404*)

9.11 TrustedContainer Interface Reference

Inheritance diagram for TrustedContainer:



Collaboration diagram for TrustedContainer:



Public Types

- typedef std::unique_ptr< TrustedContainer, CustomDeleter > uptr_t
- typedef std::unique_ptr< const TrustedContainer, CustomDeleter > uptrc_t
- using ContentType = CryptoObjectType

Public Member Functions

- virtual ContentType getObjectId (COUID ∗objectUid=nullptr) const noexcept(true)=0
- virtual ContentType getDependenceId (COUID ∗objectUid=nullptr) const noexcept(true)=0
- virtual std::size_t capacity () const noexcept(true)=0
- virtual bool isVolatile () const noexcept(true)=0
- virtual bool isObjectSession () const noexcept(true)=0
- virtual bool isObjectExportable () const noexcept(true)=0
- virtual std::size_t objectSize () const noexcept(true)=0
- virtual ContentType typeRestriction () const noexcept(true)=0
- virtual AllowedUsageFlags_t allowedUsage () const noexcept(true)=0

Protected Member Functions

- virtual ∼TrustedContainer ()=default

Additional Inherited Members

9.11.1 Detailed Description

[SWS_CRYPTO_10600] Interface definition: ⌈ Formal interface of a Trusted Container is used for saving and loading of security objects.

Note

> Actual saving and loading should be implemented by internal methods known to a trusted pair of Crypto Provider and Storage Provider.
> Each object should be uniquely identified by its type and Crypto Object Unique Identifier (COUID).
> This interface suppose that objects in the container are compressed i.e. have a minimal size optimized for saving or transferring.

⌋*(RS_CRYPTO_02004)*

9.11.2 Member Typedef Documentation

9.11.2.1 typedef std::unique_ptr<TrustedContainer, CustomDeleter> uptr_t

[SWS_CRYPTO_10601] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.11.2.2 typedef std::unique_ptr<const TrustedContainer, CustomDeleter> uptrc_t

[SWS_CRYPTO_10602] Interface definition: ⌈ Unique smart pointer of the constant interface.

⌋*(RS_CRYPTO_02404)*

9.11.2.3 using ContentType = CryptoObjectType

[SWS_CRYPTO_10603] Interface definition: ⌈ Content Type of the Trusted Container.

⌋*(RS_CRYPTO_02004)*

9.11.3 Constructor & Destructor Documentation

9.11.3.1 virtual ~TrustedContainer ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_10610] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.11.4 Member Function Documentation

9.11.4.1 virtual ContentType getObjectId (COUID ∗ *objectUid* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_10611] Interface definition: ⌈ Returns COUID and type of an object stored to this trusted container.

Parameters

`out`	*objectUid*	An optional pointer to buffer for getting COUID of an object stored in the container.

Returns

Type of the content stored in the container.

Note

If the container is empty then this method returns ContentType::NONE.
If (objectUid != nullptr), but the container is empty then the objectUid will be filled by all zeros.

⌋*(RS_CRYPTO_02004)*

9.11.4.2 virtual ContentType getDependenceId (COUID ∗ *objectUid* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_10612] Interface definition: ⌈ Returns COUID and type of an object from which current object (in the container) depends.

Parameters

`out`	*objectUid*	An optional pointer to buffer for getting COUID an object from which current object depends.

Returns

Type of an object from which current one depends or `ContentType::NONE` if it has no any dependencies.

Note

If the object doesn't depend from (or refer to) another object then the objectId will be filled by all zeros.

⌋(*RS_CRYPTO_02311*)

9.11.4.3 virtual std::size_t capacity ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10613] Interface definition: ⌈ Returns capacity of the trusted container.

Returns

Capacity of the trusted container (in bytes).

⌋(*RS_CRYPTO_02110*)

9.11.4.4 virtual bool isVolatile ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10614] Interface definition: ⌈ Returns volatility of the trusted container.

Returns

`true` if the container has a volatile nature (i.e. "temporary" or "in RAM") or `false` otherwise.

Note

A "session" object can be stored to a "volatile" container only!
A content of a "volatile" container will be destroyed together with the interface instance!

⌋(*RS_CRYPTO_02311*)

9.11.4.5 virtual bool isObjectSession ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10615] Interface definition: ⌈ Returns the "session" (or "temporary") attribute of an object stored to the container.

Returns

`true` if an object stored to the container has set the "session" attribute.

Note

A "session" object can be stored to a "volatile" container only!
If a "volatile" container keeps a non-session object then it can be saved permanently.

⌋(*RS_CRYPTO_02311*)

9.11.4.6 virtual bool isObjectExportable ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10616] Interface definition: ⌈ Returns the "exportable" attribute of an object stored to the container.

Returns

`true` if an object stored to the container has set the "exportable" attribute.

Note

The exportability of an object doesn't depend from the volatility of its container.

⌋(*RS_CRYPTO_02311*)

9.11.4.7 virtual std::size_t objectSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10617] Interface definition: ⌈ Returns size of an object stored to the trusted container.

Returns

Size of an object stored to the trusted container (in bytes).

Note

If the container is empty then this method returns 0.

⌋(*RS_CRYPTO_02311*)

9.11.4.8 virtual ContentType typeRestriction ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10618] Interface definition: ⌈ Returns content type restriction of this trusted container.

Returns

An object type of allowed content (`ContentType::NONE` means without restriction).

Note

If a container has a type restriction different from `ContentType::NONE` then only objects of the mentioned type can be saved to this container.
Volatile containers don't have any content type restrictions.

⌋*(RS_CRYPTO_02004, RS_CRYPTO_02110)*

9.11.4.9 virtual AllowedUsageFlags_t allowedUsage ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_10619] Interface definition: ⌈ Returns actual allowed key/seed usage flags defined by the key slot prototype for this Actor (application) and current content of the container.

Returns

Allowed key/seed usage flags.

Note

Volatile containers don't have any prototyped restrictions, but can have restrictions defined at run-time for a current instance of object.
A value returned by this method is bitwise AND of the common usage flags defined at run-time and the usage flags defined by the UserPermissions prototype for current Actor (application).
This method is especially useful for empty permanent prototyped containers!

⌋*(RS_CRYPTO_02008)*

9.12 UUID Struct Reference

Public Member Functions

- UUID ()
- UUID (const UUID &)=default
- UUID (UUID &&)=delete
- UUID & operator= (const UUID &)=default
- UUID & operator= (UUID &&)=delete

9.12.1 Detailed Description

[SWS_CRYPTO_10400] Interface definition: ⌈ Definition of Universally Unique Identifier (UUID) or Globally Unique Identifier (GUID) type.

Note

> This definition is just an example and can be changed to any other. Actually only its common length equal to 128 bit is principal!

⌋*(RS_CRYPTO_02404)*

9.12.2 Constructor & Destructor Documentation

9.12.2.1 UUID ( ) `[inline]`

[SWS_CRYPTO_10411] Interface definition: ⌈ Default constructor (initializes the identifier by zeros).

⌋*(RS_CRYPTO_02311)*

9.12.2.2 UUID (const UUID &) `[default]`

Default behavior of move and copy.

9.12.2.3 UUID (UUID &&) `[delete]`

Default behavior of move and copy.

9.12.3 Member Function Documentation

9.12.3.1 UUID& operator= (const UUID &) `[default]`

Default behavior of move and copy.

9.12.3.2 UUID& operator= (UUID &&) `[delete]`

Default behavior of move and copy.

9.12.4 Member Data Documentation

9.12.4.1 std::uint64_t qwordMS

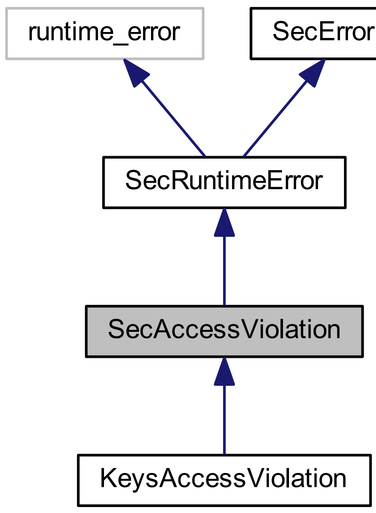
Most significant QWORD.

9.12.4.2 std::uint64_t qwordLS

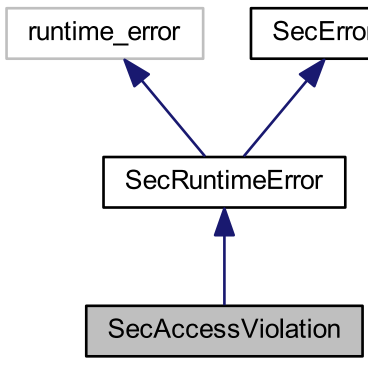
Less significant QWORD.

9.13 SecAccessViolation Interface Reference

Inheritance diagram for SecAccessViolation:



Collaboration diagram for SecAccessViolation:



Additional Inherited Members

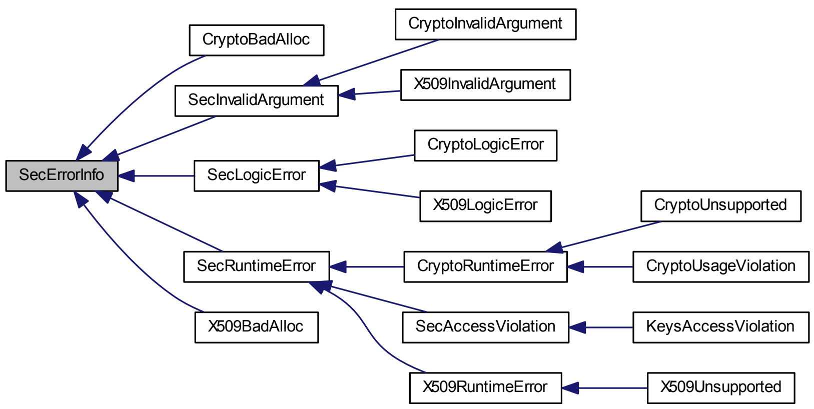
9.13.1 Detailed Description

[SWS_CRYPTO_19600] Interface definition: ⌈ An interface of the Security Access Violation exception.

⌋*(RS_CRYPTO_02310)*

9.14 SecErrorInfo Interface Reference

Inheritance diagram for SecErrorInfo:



Public Types

- typedef std::uint32_t ErrorCode_t

Public Member Functions

- virtual ∼SecErrorInfo ()=default
- virtual ErrorDomain errorDomain () const noexcept(true)=0
- virtual ErrorCode_t generalErrorCode () const noexcept(true)=0
- virtual ErrorCode_t extendedErrorCode () const noexcept(true)=0

Static Public Attributes

- static const ErrorCode_t NO_ERROR = 0

9.14.1 Detailed Description

[SWS_CRYPTO_19100] Interface definition: ⌈ A basic interface for all classes that can report details about their error state.

⌋*(RS_CRYPTO_02310)*

9.14.2 Member Typedef Documentation

9.14.2.1 typedef std::uint32_t ErrorCode_t

[SWS_CRYPTO_19101] Interface definition: ⌈ Unified Error Code type definition.

⌋*(RS_CRYPTO_02311)*

9.14.3 Constructor & Destructor Documentation

9.14.3.1 virtual ∼SecErrorInfo ( ) `[virtual], [default]`

[SWS_CRYPTO_19110] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.14.4 Member Function Documentation

9.14.4.1 virtual ErrorDomain errorDomain ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_19111] Interface definition: ⌈ Returns a domain of this error .

Returns

Error Domain of this exception.

⌋*(RS_CRYPTO_02310)*

9.14.4.2 virtual ErrorCode_t generalErrorCode ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_19112] Interface definition: ⌈ Returns a general error code.

Returns

General Error Code of this exception.

Note

It can be a code of a top-level error.

⌋*(RS_CRYPTO_02310)*

9.14.4.3 virtual ErrorCode_t extendedErrorCode ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_19113] Interface definition: ⌈ Returns an extended error code.

Returns

Extended error code of this exception.

Note

It can be a code of a low-level error.

⌋(*RS_CRYPTO_02310*)

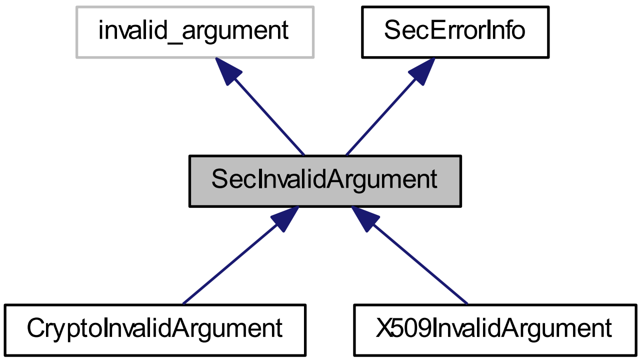
9.14.5 Member Data Documentation

9.14.5.1 const ErrorCode_t NO_ERROR = 0 `[static]`

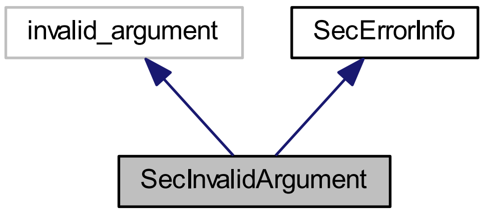
The zero value is reserved for the "No Error" state.

9.15 SecInvalidArgument Interface Reference

Inheritance diagram for SecInvalidArgument:



Collaboration diagram for SecInvalidArgument:



Additional Inherited Members

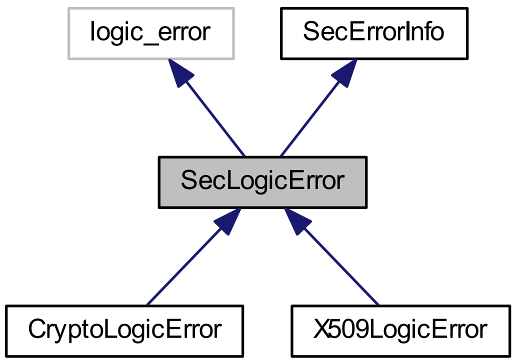
9.15.1 Detailed Description

[SWS_CRYPTO_19700] Interface definition: ⌈ An interface of the Security Invalid Argument exception

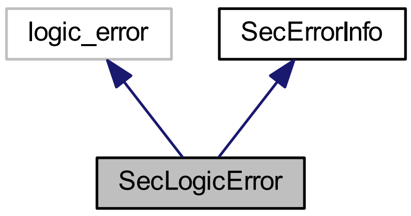
⌋*(RS_CRYPTO_02310)*

9.16 SecLogicError Interface Reference

Inheritance diagram for SecLogicError:



Collaboration diagram for SecLogicError:



Additional Inherited Members

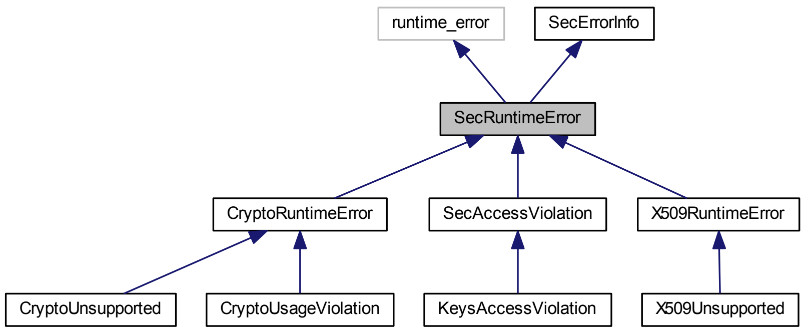
9.16.1 Detailed Description

[SWS_CRYPTO_19800] Interface definition: ⌈ A common interface for all Security Logic Error exceptions

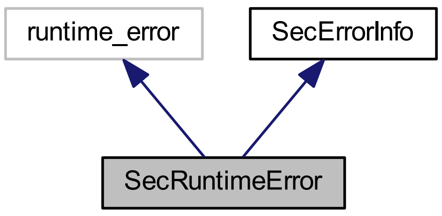
⌋(*RS_CRYPTO_02310*)

9.17 SecRuntimeError Interface Reference

Inheritance diagram for SecRuntimeError:



Collaboration diagram for SecRuntimeError:



Additional Inherited Members

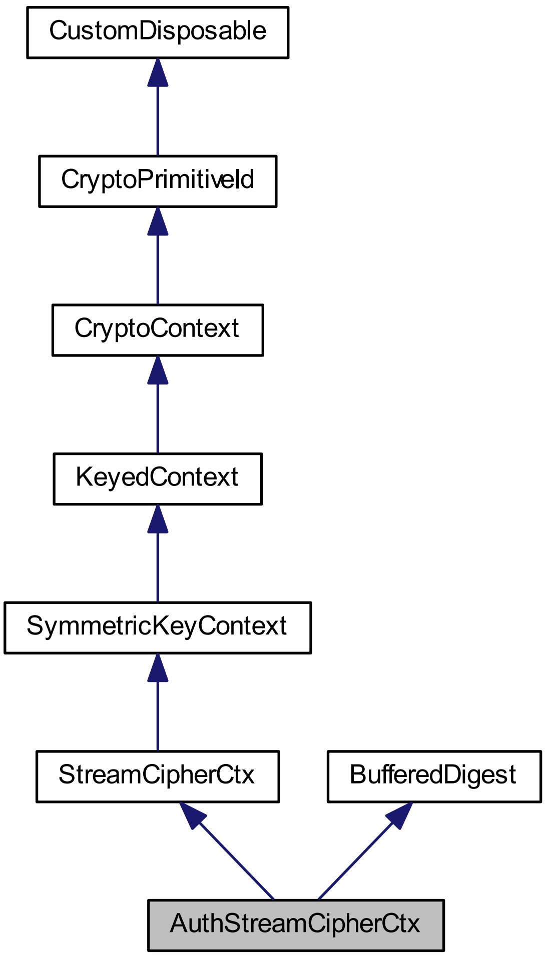
9.17.1 Detailed Description

[SWS_CRYPTO_19900] Interface definition: ⌈ A common interface for all Security Runtime Error exceptions

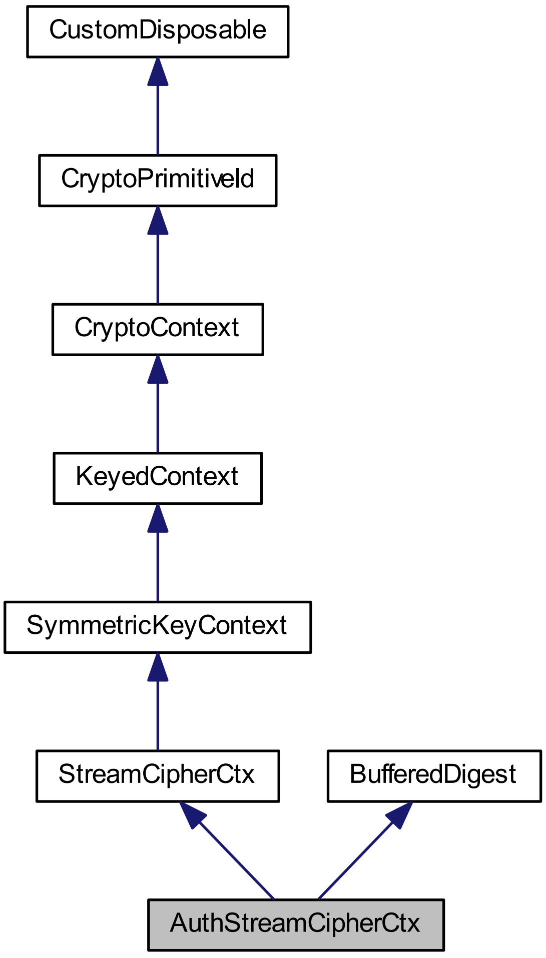
⌋*(RS_CRYPTO_02310)*

9.18 AuthStreamCipherCtx Interface Reference

Inheritance diagram for AuthStreamCipherCtx:



Collaboration diagram for AuthStreamCipherCtx:



Public Types

- typedef std::unique_ptr< AuthStreamCipherCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::uint64_t getMaxAssociatedDataSize () const noexcept(true)=0

Additional Inherited Members

9.18.1 Detailed Description

[SWS_CRYPTO_20100] Interface definition: ⌈ Generalized Authenticated Stream Cipher Context interface.

⌋*(RS_CRYPTO_02207)*

9.18.2 Member Typedef Documentation

9.18.2.1 typedef std::unique_ptr<AuthStreamCipherCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_20101] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.18.3 Member Function Documentation

9.18.3.1 virtual std::uint64_t getMaxAssociatedDataSize ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20110] Interface definition: ⌈ Gets maximal supported size of associated public data.

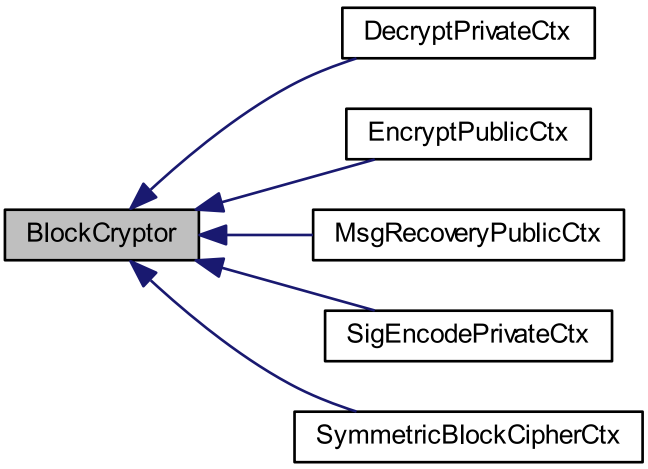
Returns

Maximal supported size of associated public data in bytes.

⌋*(RS_CRYPTO_02309)*

9.19 BlockCryptor Interface Reference

Inheritance diagram for BlockCryptor:



Public Member Functions

- virtual bool isEncryption () const noexcept(true)=0
- bool isMaxInputOnly () const noexcept(true)
- bool isMaxOutputOnly () const noexcept(true)
- virtual std::size_t maxInputSize (bool suppressPadding=false) const noexcept(true)=0
- virtual std::size_t maxOutputSize (bool suppressPadding=false) const noexcept(true)=0
- virtual std::size_t processBlock (WritableMemRegion out, ReadOnlyMemRegion in, bool suppressPadding=false) const noexcept(false)=0
- template<typename Alloc >
  void processBlock (std::vector< byte_t, Alloc > &out, ReadOnlyMemRegion in, bool suppressPadding=false) const noexcept(false)

Protected Member Functions

- virtual ~BlockCryptor ()=default

9.19.1 Detailed Description

[SWS_CRYPTO_20200] Interface definition: ⌈ General interface for stateless encryption / decryption of a single data block with padding.

Note

> The Block Cryptor context should include a defenition of a padding scheme applicable by default.
> Use non-default value of argument `suppressPadding` only if you know exactly what you are doing!

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202)*

9.19.2 Constructor & Destructor Documentation

9.19.2.1 virtual ~BlockCryptor ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_20210] Interface definition: ⌈ Virtual destructor.

⌋*(RS_CRYPTO_02311)*

9.19.3 Member Function Documentation

9.19.3.1 virtual bool isEncryption ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20211] Interface definition: ⌈ Returns target transformation of the current configuration of the Block Cryptor context.

Returns

> `true` if the Block Cryptor context is configured for encryption and `false` for decryption.

⌋*(RS_CRYPTO_02309)*

9.19.3.2 bool isMaxInputOnly ( ) const `[inline]`, `[noexcept]`

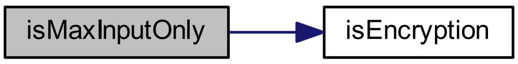
[SWS_CRYPTO_20212] Interface definition: ⌈ Indicates that the currently configured transformation accepts only complete blocks of input data.

Returns

true if only the maximum size of input data is required by the transformation.

⌋*(RS_CRYPTO_02309)*

Here is the call graph for this function:



9.19.3.3 bool isMaxOutputOnly ( ) const [inline], [noexcept]

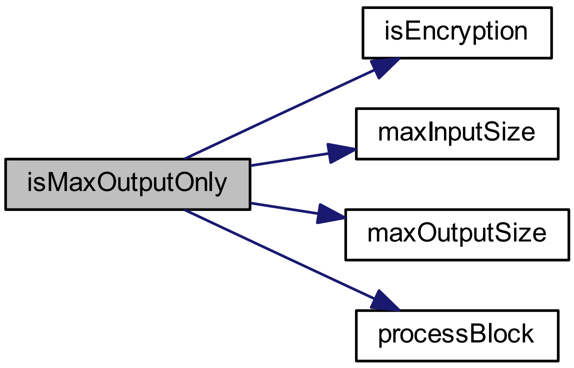
[SWS_CRYPTO_20213] Interface definition: ⌈ Indicates that the currently configured transformation can produce only complete blocks of output data.

Returns

true if only the maximum size of output data can be produced by the transformation.

⌋*(RS_CRYPTO_02309)*

Here is the call graph for this function:



9.19.3.4 virtual std::size_t maxInputSize (bool *suppressPadding* = `false`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20214] Interface definition: ⌈ Returns maximum expected size of the input data block.

Parameters

`in`	*suppressPadding*	If `true` then the method calculates the size for the case when whole space of the plain data block is used for a payload only.

Returns

Maximum size of the input data block in bytes.

Note

If `(isEncryption() == false)` then a value returned by this method is independent from the `suppressPadding` argument and will be equal to the block size.

⌋(*RS_CRYPTO_02309*)

9.19.3.5 virtual std::size_t maxOutputSize (bool *suppressPadding* = `false`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20215] Interface definition: ⌈ Returns maximum possible size of the output data block.

Parameters

`in`	*suppressPadding*	If `true` then the method calculates the size for the case when whole space of the plain data block is used for a payload only.

Returns

Maximum size of the output data block in bytes.

Note

If `(isEncryption() == true)` then a value returned by this method is independent from the `suppressPadding` argument and will be equal to the block size.

⌋(*RS_CRYPTO_02309*)

9.19.3.6 virtual std::size_t processBlock (WritableMemRegion *out*, ReadOnlyMemRegion *in*, bool *suppressPadding* = `false`) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20216] Interface definition: ⌈ Process (encrypt / decrypt) an input block according to the cryptor configuration.

Parameters

`out`	*out*	An output buffer.
`in`	*in*	An input data block.
`in`	*suppressPadding*	If `true` then the method doesn't apply the padding, but the payload should fill the whole block of the plain data.

Returns

Actual size of output data (it always `<= out.size()`) or 0 if the input data block has incorrect content.

Note

Encryption with `(suppressPadding == true)` expects that: `in.size() == maxInputSize(true) && out.size() >= maxOutputSize(true)`.
Encryption with `(suppressPadding == false)` expects that: `in.size() <= maxInputSize(false) && in.size() > 0 && out.size() >= maxOutputSize(false)`.
Decryption expects that: `in.size() == maxInputSize() && out.size() >= maxOutputSize(suppressPadding)`.
The case `(out.size() < maxOutputSize())` should be used with caution, only if you are strictly certain about the size of the output data!
In case of `(suppressPadding == true)` the actual size of plain text should be equal to full size of the plain data block (defined by the algorithm)!

Exceptions

CryptoInvalidArgument	if the mentioned above rules about in/out sizes are violated or the `out.size()` is not enough to store the result.
CryptoLogicError	if the context was not initialized by a key value.

⌋(*RS_CRYPTO_02311*)

9.19.3.7 void processBlock (std::vector< byte_t, Alloc > & *out*, ReadOnlyMemRegion *in*, bool *suppressPadding* = `false`) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_20217] Interface definition: ⌈ Process (encrypt / decrypt) an input block according to the cryptor configuration.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*out*	A managed container for output block.
`in`	*in*	An input data block.
`in`	*suppressPadding*	If `true` then the method doesn't apply the padding, but the payload should fill the whole block of the plain data.

Note

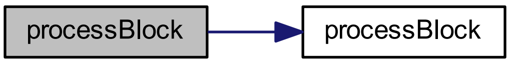
This method sets the size of the output container according to actually saved value!
Encryption with `(suppressPadding == true)` expects that: `in.size() == maxInputSize(true) && out.capacity() >= maxOutputSize(true)`.
Encryption with `(suppressPadding == false)` expects that: `in.size() <= maxInputSize(false) && in.size() > 0 && out.capacity() >= maxOutputSize(false)`.
Decryption expects that: `in.size() == maxInputSize() && out.capacity() >= maxOutputSize(suppressPadding)`.
The case `(out.capacity() < maxOutputSize())` should be used with caution, only if you are strictly certain about the size of the output data!
In case of `(suppressPadding == true)` the actual size of plain text should be equal to full size of the plain data block (defined by the algorithm)!

Exceptions

CryptoInvalidArgument	if the mentioned above rules about in/out sizes are violated or the `out.capacity()` is not enough to store the result.
CryptoLogicError	if the context was not initialized by a key value.

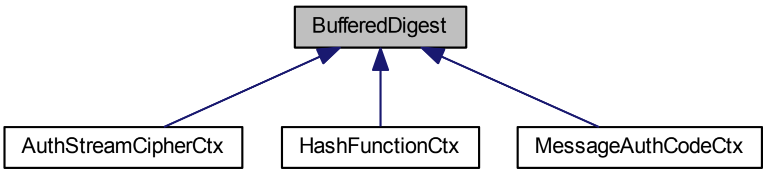
⌋(*RS_CRYPTO_02311*)

Here is the call graph for this function:



9.20 BufferedDigest Interface Reference

Inheritance diagram for BufferedDigest:



Public Member Functions

- virtual std::size_t digestSize () const noexcept(true)=0
- virtual std::size_t bufferSize () const noexcept(true)=0
- virtual void start (ReadOnlyMemRegion ∗nonce=nullptr) noexcept(false)=0
- virtual void start (const SecretSeed &nonce) noexcept(false)=0
- virtual void update (const KeyMaterial &in) noexcept(false)=0
- virtual void update (ReadOnlyMemRegion in) noexcept(false)=0
- virtual void update (byte_t in) noexcept(false)=0
- virtual Signature::uptrc_t finish (bool makeSignatureObject=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual std::size_t getDigest (WritableMemRegion output, std::size_t offset=0) const noexcept(false)=0

- template<typename Alloc >
  void getDigest (std::vector< byte_t, Alloc > &output, std::size_t offset=0) const noexcept(false)
- virtual bool compare (ReadOnlyMemRegion expected, std::size_t offset=0) const noexcept(false)=0

Protected Member Functions

- virtual ~BufferedDigest ()=default

9.20.1 Detailed Description

[SWS_CRYPTO_20300] Interface definition: ⌈ General interface for buffered computation of a digest (MAC/HMAC/hash)

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02205)*

9.20.2 Constructor & Destructor Documentation

9.20.2.1 virtual ~BufferedDigest ( ) [protected], [virtual], [default]

[SWS_CRYPTO_20310] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.20.3 Member Function Documentation

9.20.3.1 virtual std::size_t digestSize ( ) const [pure virtual], [noexcept]

[SWS_CRYPTO_20311] Interface definition: ⌈ Getter of the output digest size.

Returns

Size of the full output from this digest-function in bytes.

⌋*(RS_CRYPTO_02309)*

9.20.3.2 virtual std::size_t bufferSize ( ) const [pure virtual], [noexcept]

[SWS_CRYPTO_20312] Interface definition: ⌈ Getter of internal buffer size.

Returns

Size of the internal buffer in bytes (or block size).

Note

It is an informative method intended only for optimization of this interface usage.

⌋*(RS_CRYPTO_02309, RS_CRYPTO_02404)*

9.20.3.3 virtual void start (ReadOnlyMemRegion ∗ *nonce* = `nullptr`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20313] Interface definition: ⌈ Start the digest calculation for a new message.

Parameters

in	*nonce*	An optional pointer to nonce (or initialization vector) value.

Exceptions

CryptoInvalidArgument	if provided nonce has unsupported length (i.e. it is not enough for the initialization).
CryptoLogicError	if the context was not initialized by a key value (for keyed digest only).

⌋*(RS_CRYPTO_02302)*

9.20.3.4 virtual void start (const SecretSeed & *nonce*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20314] Interface definition: ⌈ Start the digest calculation for a new message.

Parameters

in	*nonce*	A reference to nonce (or initialization vector) value.

Exceptions

CryptoInvalidArgument	if provided nonce has unsupported length (i.e. it is not enough for the initialization).
CryptoLogicError	if the context was not initialized by a key value (for keyed digest only).
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `SecretSeed` object.

⌋(*RS_CRYPTO_02302*)

9.20.3.5 virtual void update (const KeyMaterial & *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20315] Interface definition: ⌈ Update the digest calculation context by a new part of the message.

Parameters

`in`	*in*	A part of input message that should be processed.

Exceptions

CryptoLogicError	if the context was not initialized by a call of the `start()` method.

⌋(*RS_CRYPTO_02302*)

9.20.3.6 virtual void update (ReadOnlyMemRegion *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20316] Interface definition: ⌈ Update the digest calculation context by a new part of the message.

Parameters

`in`	*in*	A part of input message that should be processed.

Exceptions

CryptoLogicError	if the context was not initialized by a call of the `start()` method.

⌋(*RS_CRYPTO_02302*)

9.20.3.7 virtual void update (byte_t *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20317] Interface definition: ⌈ Update the digest calculation context by a new part of the message.

Parameters

`in`	*in*	A byte value that is a part of input message.

Note

This method is convenient for processing of constant tags.

Exceptions

CryptoLogicError	if the context was not initialized by a call of the `start()` method.

⌋(*RS_CRYPTO_02302*)

9.20.3.8 virtual Signature::uptrc_t finish (bool *makeSignatureObject* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20318] Interface definition: ⌈ Finish the digest calculation and optionally produce the "signature" object.

Parameters

`in`	*makeSignatureObject*	If this argument is `true` then the method will also produce the signature object.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to created signature object, if `(makeSignatureObject == true)` or `nullptr` if `(makeSignatureObject == false)`.

Note

Only after call of this method the digest can be signed, verified, extracted or compared!
If the signature object produced by a plain hash-function then the dependence COUID of the "signature" should be set to COUID of domain parameters used by this context, but the "hash algorithm ID" field of the "signature" should be set according to own algorithm ID (i.e. equal to `CryptoPrimitiveId::getPrimitiveId()`).
If the signature object produced by a keyed MAC/HMAC/AE/AEAD algorithm then the dependence COUID of the "signature" should be set to COUID of used symmetric key, but the "hash algorithm ID" field of the "signature" should be set to `ALGID_NONE` (0).

Exceptions

CryptoLogicError	if the digest calculation was not initiated by a call of the `start()` method.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.
CryptoUsageViolation	if the buffered digest belongs to a MAC/HMAC context initialized by a key without `ALLOW_SIGNATURE` permission, but `(makeSignatureObject == true)`.

⌋*(RS_CRYPTO_02302, RS_CRYPTO_02404)*

9.20.3.9 virtual std::size_t getDigest (WritableMemRegion *output*, std::size_t *offset = 0*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20319] Interface definition: ⌈ Gets requested part of calculated digest to existing memory buffer.

Parameters

`out`	*output*	An output buffer for storing the requested digest fragment (or fully).

in	*offset*	Position of first byte of digest that should be placed to the output buffer.

Returns

Number of digest bytes really stored to the output buffer (they are always <= `output.size()` and denoted below as `return_size`).

Note

Entire digest value is kept in the context up to next call `start()`, therefore any its part can be extracted again or verified.
If `(full_digest_size <= offset)` then `return_size = 0` bytes;
else `return_size = min(output.size(), (full_digest_size - offset))` bytes.

Exceptions

CryptoLogicError	if the digest calculation was not finished by a call of the `finish()` method.
CryptoUsageViolation	if the buffered digest is part of MAC/HMAC context initialized by a key without `ALLOW_SIGNATURE` permission.

⌋(*RS_CRYPTO_02404*)

9.20.3.10 void getDigest (std::vector< byte_t, Alloc > & *output*, std::size_t *offset* = *0*) const [inline], [noexcept]

[SWS_CRYPTO_20320] Interface definition: ⌈ Gets requested part of calculated digest to pre-reserved managed container.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

out	*output*	A managed container for storing the requested digest fragment (or fully).
in	*offset*	Position of first byte of digest that should be placed to the output buffer.

Note

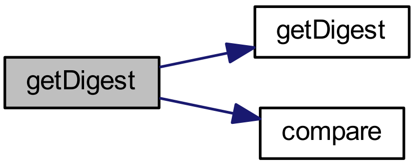
This method sets the size of the output container according to actually saved value.
Entire digest value is kept in the context up to next call `start()`, therefore any its part can be extracted again or verified.
If `(full_digest_size <= offset)` then `return_size = 0` bytes;
else `return_size = min(output.capacity(), (full_digest_size - offset))` bytes.

Exceptions

CryptoLogicError	if the digest calculation was not finished by a call of the `finish()` method.
CryptoUsageViolation	if the buffered digest is part of MAC/HMAC context initialized by a key without `ALLOW_SIGNATURE` permission.

⌋*(RS_CRYPTO_02404)*

Here is the call graph for this function:



9.20.3.11 virtual bool compare (ReadOnlyMemRegion *expected,* std::size_t *offset = 0*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20321] Interface definition: ⌈ Compares calculated digest against an expected value.

Parameters

`in`	*expected*	A memory region containing an expected digest value.
`in`	*offset*	Position of first byte in calculated digest for the comparison starting.

Returns

true if the expected bytes sequence is identical to first bytes of calculated digest.

Note

Entire digest value is kept in the context up to next call start(), therefore any its part can be verified again or extracted.
If (full_digest_size <= offset) || (expected.size() == 0) then return false;
else comparison_size = min(expected.size(), (full_digest_size - offset)) bytes.

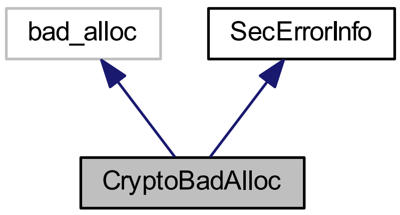
Exceptions

CryptoLogicError	if the digest calculation was not finished by a call of the finish() method.
CryptoRuntimeError	if the buffered digest is part of MAC/HMAC context initialized by a key without ALLOW_SIGNATURE permission, but actual size of requested digest is less than 8 bytes (it is a protection from the brute-force attack).

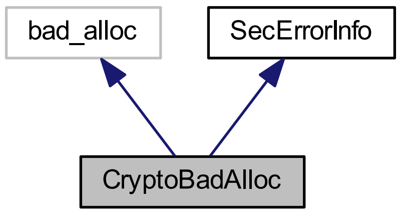
⌋(*RS_CRYPTO_02404*)

9.21 CryptoBadAlloc Interface Reference

Inheritance diagram for CryptoBadAlloc:



Collaboration diagram for CryptoBadAlloc:



Additional Inherited Members

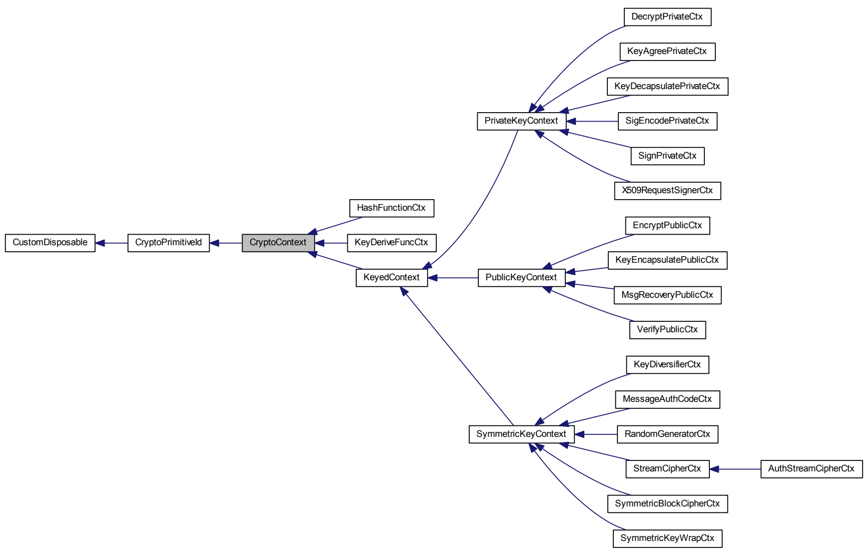
9.21.1 Detailed Description

[SWS_CRYPTO_29400] Interface definition: ⌈ An interface for the Crypto Bad Allocation exception

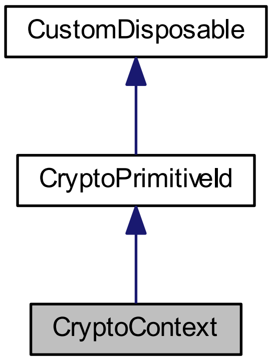
⌋*(RS_CRYPTO_02310)*

9.22 CryptoContext Interface Reference

Inheritance diagram for CryptoContext:



Collaboration diagram for CryptoContext:



Public Member Functions

- virtual bool isKeyedContext () const noexcept(true)=0
- virtual bool isInitialized () const noexcept(true)=0
- virtual void clear () noexcept(true)=0
- virtual void setParameters (DomainParameters::sptrc_t parameters) noexcept(false)=0

Additional Inherited Members

9.22.1 Detailed Description

[SWS_CRYPTO_20400] Interface definition: ⌈ A common interface of a mutable cryptographic context, i.e. that is not binded to a single crypto object.

⌋*(RS_CRYPTO_02311)*

9.22.2 Member Function Documentation

9.22.2.1 virtual bool isKeyedContext ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20411] Interface definition: ⌈ Checks if the crypto context requires initialization by a key value.

Returns

`true` if the crypto context requires initialization by a key value.

⌋*(RS_CRYPTO_02309)*

9.22.2.2 virtual bool isInitialized ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20412] Interface definition: ⌈ Checks if the crypto context is already initialized (by all required values, incl.: domain parameters, key value, IV/seed, etc.) and ready to use.

Returns

`true` if the crypto context is already initialized and ready to use.

⌋*(RS_CRYPTO_02309)*

9.22.2.3 virtual void clear ( ) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20413] Interface definition: ⌈ Clear context after previous calculations (from key material and/or temporary data).

⌋*(RS_CRYPTO_02311)*

9.22.2.4 virtual void setParameters (DomainParameters::sptrc_t *parameters*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20414] Interface definition: ⌈ Clear context after previous calculations (from key material and/or temporary data).

Parameters

`in`	*parameters*	Shared pointer to "domain parameters" applicable for this cryptographic context (algorithm).

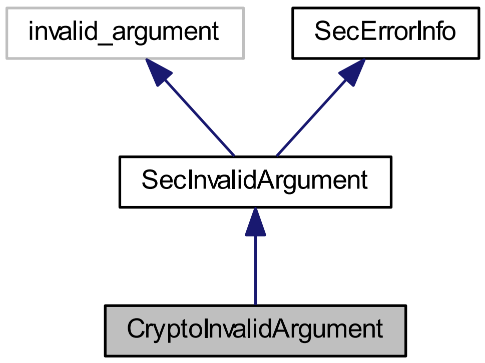
Exceptions

CryptoRuntimeError	if provided parameters are not applicable to current cryptographic context (algorithm).

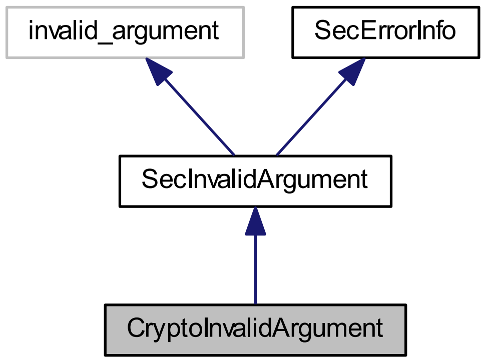
⌋*(RS_CRYPTO_02108)*

9.23 CryptoInvalidArgument Interface Reference

Inheritance diagram for CryptoInvalidArgument:



Collaboration diagram for CryptoInvalidArgument:



Additional Inherited Members

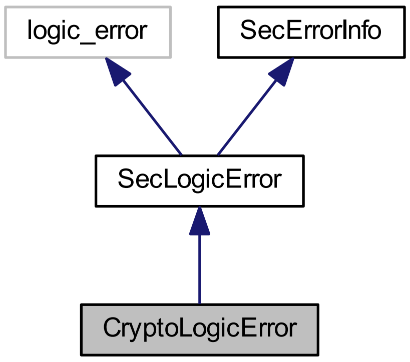
9.23.1 Detailed Description

[SWS_CRYPTO_29500] Interface definition: ⌈ An interface of the Crypto Invalid Argument exception

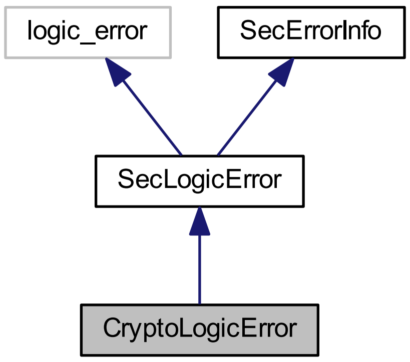
⌋*(RS_CRYPTO_02310)*

9.24 CryptoLogicError Interface Reference

Inheritance diagram for CryptoLogicError:



Collaboration diagram for CryptoLogicError:



Additional Inherited Members

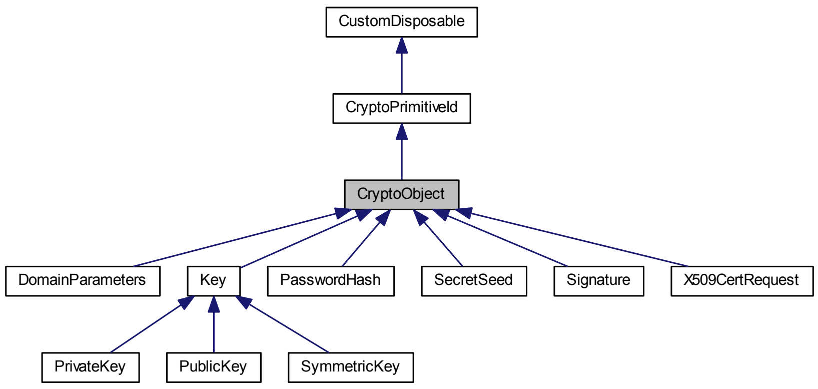
9.24.1 Detailed Description

[SWS_CRYPTO_29600] Interface definition: ⌈ A common interface for all Crypto Logic Error exceptions

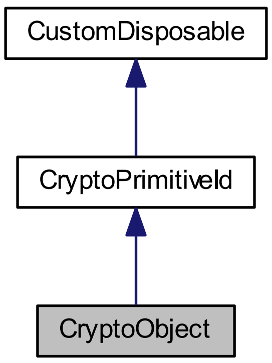
⌋*(RS_CRYPTO_02310)*

9.25 CryptoObject Interface Reference

Inheritance diagram for CryptoObject:



Collaboration diagram for CryptoObject:



Public Types

- typedef std::unique_ptr< CryptoObject, CustomDeleter > uptr_t
- typedef std::unique_ptr< const CryptoObject, CustomDeleter > uptrc_t
- using Type = CryptoObjectType

Public Member Functions

- virtual Type getObjectType () const noexcept(true)=0
- virtual bool isSession () const noexcept(true)=0
- virtual bool isExportable () const noexcept(true)=0
- virtual bool getObjectId (COUID ∗objectId=nullptr) const noexcept(true)=0
- virtual Type hasDependence (COUID ∗objectId=nullptr) const noexcept(true)=0
- virtual std::size_t storageSize () const noexcept(true)=0
- virtual void save (TrustedContainer &container) const noexcept(false)=0

Additional Inherited Members

9.25.1 Detailed Description

[SWS_CRYPTO_20500] Interface definition: ⌈ A common interface for all cryptograhic objects recognizable by the Crypto Provider.

Note

> This interface (or any its derivative) represents a non-mutable (after completion) object loadable to a temporary transformation context.

⌋*(RS_CRYPTO_02311)*

9.25.2 Member Typedef Documentation

9.25.2.1 typedef std::unique_ptr<CryptoObject, CustomDeleter> uptr_t

[SWS_CRYPTO_20501] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.25.2.2 typedef std::unique_ptr<const CryptoObject, CustomDeleter> uptrc_t

[SWS_CRYPTO_20502] Interface definition: ⌈ Unique smart pointer of the constant interface.

⌋*(RS_CRYPTO_02404)*

9.25.2.3 using Type = CryptoObjectType

[SWS_CRYPTO_20503] Interface definition: ⌈ Enumeration of all types of crypto objects.

⌋*(RS_CRYPTO_02311)*

9.25.3 Member Function Documentation

9.25.3.1 virtual Type getObjectType ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20511] Interface definition: ⌈ Returns the type of this object.

Returns

One of object types except `Type::UNKNOWN`.

⌋*(RS_CRYPTO_02311)*

9.25.3.2 virtual bool isSession ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20512] Interface definition: ⌈ Returns the "session" (or "temporary") attribute of the object.

Returns

`true` if the object is temporay (i.e. its life time is limited by the current session only).

Note

A temporary object cannot be saved to a non-volatile trusted container!
A temporary object will be securely destroyed together with this interface instance!
A non-session object must have an assigned COUID (see `getObjectId()`).

⌋*(RS_CRYPTO_02003)*

9.25.3.3 virtual bool isExportable ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20513] Interface definition: ⌈ Returns the "exportable" attribute of the object.

Returns

`true` if the object is exportable (i.e. if it can be exported outside the trusted environment of the Crypto Provider).

Note

An exportable object must have an assigned COUID (see `getObjectId()`).

⌋(*RS_CRYPTO_02113*)

Implemented in Key.

9.25.3.4 virtual bool getObjectId (COUID ∗ *objectId* = `nullptr`) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20514] Interface definition: ⌈ Returns the object's UID if it is assigned to the object.

Parameters

`out`	*objectId*	Optional pointer to a buffer for the object's UID saving.

Returns

`true` if the object has assigned COUID.

Note

An object that has no an assigned COUID cannot be (securely) serialized / exported or saved to a non-volatile container.
An object should not have a COUID if it is session and non-exportable simultaneously or if it is incomplete yet (last is applicable for domain parameters only).
A few related objects of different types can share a single COUID (e.g. private and public keys), but a combination of COUID and object type must be unique always!

⌋(*RS_CRYPTO_02005*)

9.25.3.5 virtual Type hasDependence (COUID ∗ *objectId* = `nullptr`) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20515] Interface definition: ⌈ Returns an identifier of an object from which depends the current one.

Parameters

`out`	*objectId*	Optional pointer to a buffer for the target object UID.

Returns

Target object type for existing dependence or `Type::UNKNOWN` if the current object doesn't depend from any other one.

Note

> If `(objectId != nullptr)` but the current object has no dependence from other one then the `objectId` should be filled by zeros.
> For signatures objects this method **must** return a reference to correspondent signature verification public key!
> For keys objects this method should return a reference to domain parameters.
> For domain parameters object this method (optionally) can return a reference to another domain parameters required for this one.

⌋*(RS_CRYPTO_02005)*

9.25.3.6 virtual std::size_t storageSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20516] Interface definition: ⌈ Returns a storage size of the object.

Returns

> Size in bytes of the objects required for its storage.

⌋*(RS_CRYPTO_02309)*

9.25.3.7 virtual void save (TrustedContainer & *container*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20517] Interface definition: ⌈ Saves itself to provided trusted container.

Parameters

`in`	*container*	A target trusted container.

Note

> An object can be saved to a container only if the container is empty and has enough capacity.
> The save operation is allowed to an empty container only, i.e. any updates of objects are prohibited!
> Only a non-session/non-temporary and completed object (i.e. that have a COUID) can be saved!
> Only a single instance of an object is allowed in the persistent storage! Any object is uniquely identified by the combination of its COUID and type.

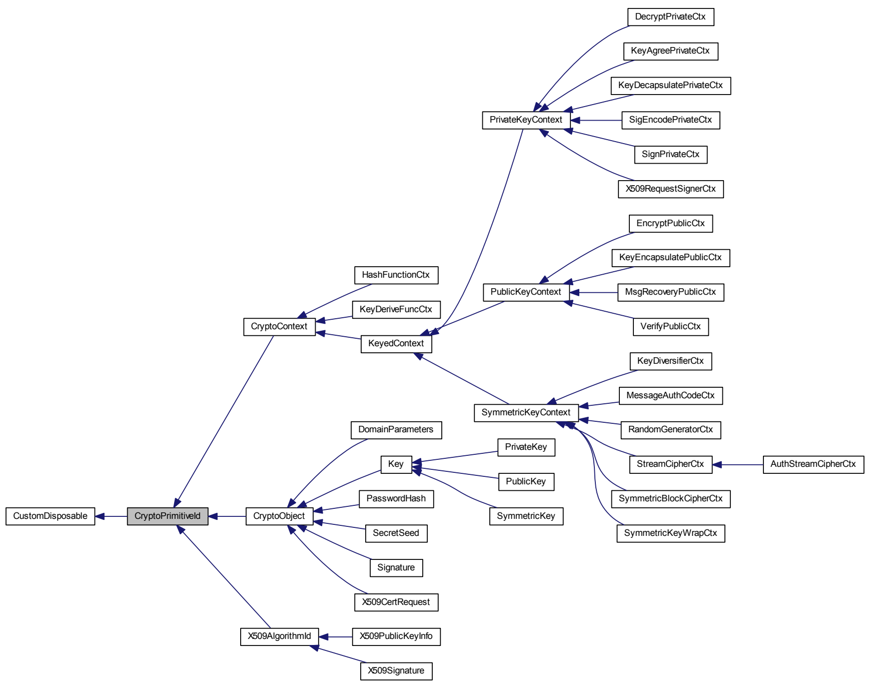
Exceptions

SecRuntimeError	if any of mentioned below conditions is satisfied: • the target container is not empty; • an object with same COUID and type already exists in the Key Storage (if the container is non-volatile); • an object is "session", but the container is non-volatile or doesn't satisfy the slot restrictions.

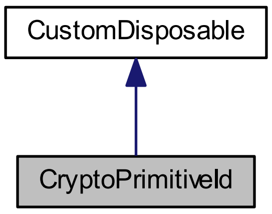
⌋(*RS_CRYPTO_02004*)

9.26 CryptoPrimitiveId Interface Reference

Inheritance diagram for CryptoPrimitiveId:



Collaboration diagram for CryptoPrimitiveId:



Public Types

- using AlgId_t = CryptoAlgId_t

Public Member Functions

- virtual const std::string & getPrimitiveName () const noexcept(true)=0
- virtual AlgId_t getPrimitiveId () const noexcept(true)=0
- virtual Category getCategory () const noexcept(true)=0
- virtual CryptoProvider & myProvider () const noexcept(true)=0

Additional Inherited Members

9.26.1 Detailed Description

[SWS_CRYPTO_20600] Interface definition: ⌈ Common interface for identification of all Crypto Primitives and their keys & parameters.

⌋*(RS_CRYPTO_02311)*

9.26.2 Member Typedef Documentation

9.26.2.1 using AlgId_t = CryptoAlgId_t

[SWS_CRYPTO_20601] Interface definition: ⌈ Type definition of vendor specific binary Crypto Primitive ID.

⌋*(RS_CRYPTO_02311)*

9.26.3 Member Enumeration Documentation

9.26.3.1 enum Category : std::uint8_t `[strong]`

[SWS_CRYPTO_20602] Interface definition: ⌈ Enumeration of categories of all supported crypto primitives.

⌋*(RS_CRYPTO_02311)*

Enumerator

UNKNOWN A value reserved for erroneous situations.
GENERIC_SYMMETRIC_KEY Generic set of SymmetricKey primitives, applicable to key objects only!
GENERIC_ASYMMETRIC_DLP Generic set of PublicKey / PrivateKey primitives based on the Disrete Logarifm Problem (**DLP**), applicable to key objects only!
GENERIC_ASYMMETRIC_IFP Generic set of PublicKey / PrivateKey primitives based on the Integer Factoring Problem (**IFP**), applicable to key objects only!
HASH_FUNCTION Keyless `HashFunctionCtx` primitives.
KEY_DERIVATION_FUNCTION Keyless `KeyDeriveFuncCtx` primitives.
SYMMETRIC_BLOCK_CIPHER Symmetric `SymmetricBlockCipherCtx` primitives.
SYMMETRIC_STREAM_CIPHER Symmetric `StreamCipherCtx` primitives.
SYMMETRIC_AUTHENTICATION Symmetric `MessageAuthCodeCtx` primitives.
AUTHENTIC_STREAM_CIPHER Symmetric `AuthStreamCipherCtx` primitives.
KEY_DEVERSIFICATION Symmetric `KeyDiversifierCtx` primitives.
SYMMETRIC_KEY_WRAP Symmetric `SymmetricKeyWrapCtx` primitives.
RANDOM_GENERATOR `RandomGeneratorCtx` primitives
KEY_AGREEMENT_DLP Asymmetric key agreement primitives, based on the DLP (`KeyAgreePrivateCtx`).
DIGITAL_SIGNATURE_DLP Asymmetric signature primitives, based on the DLP (`SignPrivateCtx` / `VerifyPublicCtx`).
SIGNATURE_ENCODER_IFP Asymmetric signature encoding primitives with message recovery, based on the IFP (`SigEncodePrivateCtx` / `MsgRecoveryPublicCtx`).
ASYMMETRIC_CIPHER_IFP Asymmetric cipher primitives, based on the IFP (`EncryptPublicCtx` / `DecryptPrivateCtx`).
KEY_ENCAPSULATION_IFP Asymmetric key encapsulation primitives, based on the IFP (`KeyEncapsulatePublicCtx` / `KeyDecapsulatePrivateCtx`).

9.26.4 Member Function Documentation

9.26.4.1 virtual const std::string& getPrimitiveName ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20611] Interface definition: ⌈ Gets a unified name of the primitive.

Returns

A constant reference to string containing a unified name of the crypto primitive.

Note

The crypto primitive name can be fully or partially specified (see "Crypto Primitives Naming Convention" for more details).

⌋*(RS_CRYPTO_02308)*

9.26.4.2 virtual AlgId_t getPrimitiveId ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20612] Interface definition: ⌈ Gets vendor specific ID of the primitive.

Returns

The binary Crypto Primitive ID.

⌋*(RS_CRYPTO_02309)*

9.26.4.3 virtual Category getCategory ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20613] Interface definition: ⌈ Gets the category of the primitive.

Returns

The category of the primitive.

⌋*(RS_CRYPTO_02309)*

9.26.4.4 virtual CryptoProvider& myProvider ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20614] Interface definition: ⌈ Gets a reference to Crypto Provider of this primitive.

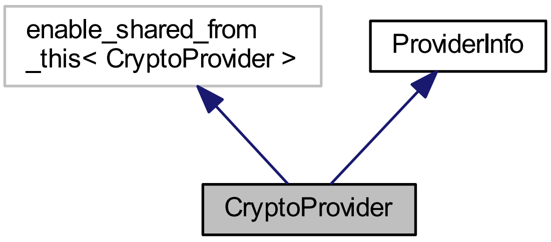
Returns

A reference to Crypto Provider instance that provides this primitive.

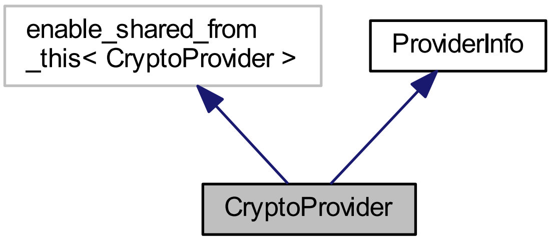
⌋(*RS_CRYPTO_02401*)

9.27 CryptoProvider Interface Reference

Inheritance diagram for CryptoProvider:



Collaboration diagram for CryptoProvider:



Public Types

- typedef std::shared_ptr< CryptoProvider > sptr_t
- using ObjectType = CryptoObject::Type
- using AlgId_t = CryptoPrimitiveId::AlgId_t
- using ContainedContextsList_t = std::vector< std::pair< AlgId_t, bool > >

- using ContainedObjectsList_t = std::vector< std::pair< AlgId_t, ObjectType > >
- using ContextReservationMap_t = std::vector< ContainedContextsList_t >
- using ObjectReservationMap_t = std::vector< ContainedObjectsList_t >

Public Member Functions

- virtual ~CryptoProvider ()=default
- virtual AlgId_t convertToAlgId (const std::string &primitiveName) const noexcept(true)=0
- virtual std::string convertToAlgName (AlgId_t algId) const noexcept(false)=0
- virtual void reserveContexts (const ContextReservationMap_t &reservationMap) noexcept(false)=0
- virtual void reserveObjects (const ObjectReservationMap_t &reservationMap) noexcept(false)=0
- virtual void reserveContexts (std::size_t quantity) noexcept(false)=0
- virtual void reserveObjects (std::size_t quantity) noexcept(false)=0
- virtual void enterRealTimeMode () noexcept(true)=0
- virtual void leaveRealTimeMode () noexcept(true)=0
- virtual DomainParameters::sptr_t allocDomainParameters (AlgId_t algId, bool isSession=false, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual DomainParameters::sptrc_t knownDomainParameters (const std::string &oidName, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual SymmetricKey::uptrc_t generateSymmetricKey (AlgId_t algId, Key::Usage_t allowedUsage, bool isSession=true, bool isExportable=false, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual PrivateKey::uptrc_t generatePrivateKey (AlgId_t algId, Key::Usage_t allowedUsage, bool isSession=false, bool isExportable=false, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual SecretSeed::uptrc_t generateSeed (AlgId_t algId, SecretSeed::Usage_t allowedUsage, bool isSession=false, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual std::size_t getSerializedSize (ObjectType objectType, AlgId_t algId, Serializable::FormatId_t formatId=Serializable::FORMAT_DEFAULT) const noexcept(false)=0

- virtual std::size_t getStorageSize (ObjectType objectType, AlgId_t algId) const noexcept(false)=0
- virtual TrustedContainer::uptr_t allocVolatileContainer (std::size_t capacity=0) noexcept(false)=0
- virtual TrustedContainer::uptr_t allocVolatileContainer (const ContainedObjectsList_t &objectsList) noexcept(false)=0
- virtual std::size_t exportSecuredObject (const CryptoObject &object, SymmetricKeyWrapCtx &transportContext, WritableMemRegion ∗serialized=nullptr) noexcept(false)=0
- virtual std::size_t exportSecuredObject (const TrustedContainer &container, SymmetricKeyWrapCtx &transportContext, WritableMemRegion ∗serialized=nullptr) noexcept(false)=0
- virtual void importSecuredObject (TrustedContainer &container, ReadOnlyMemRegion serialized, SymmetricKeyWrapCtx &transportContext, bool isExportable=false, ObjectType expectedObject=ObjectType::UNKNOWN) noexcept(false)=0
- virtual std::size_t exportPublicObject (const TrustedContainer &container, WritableMemRegion ∗serialized=nullptr, Serializable::FormatId_t formatId=Serializable::FORMAT_DEFAULT) noexcept(false)=0
- virtual void importPublicObject (TrustedContainer &container, ReadOnlyMemRegion &serialized, ObjectType expectedObject=ObjectType::UNKNOWN) noexcept(false)=0
- virtual Key::uptrc_t loadKey (const TrustedContainer &container, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual SecretSeed::uptrc_t loadSeed (const TrustedContainer &container, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual DomainParameters::uptrc_t loadDomainParameters (const TrustedContainer &container, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual PasswordCacheCtx::uptr_t allocPasswordCache (std::size_t maximalLength, std::size_t requiredLength, unsigned requiredComplexity, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual PasswordHash::uptr_t loadPasswordHash (const TrustedContainer &container, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual PasswordHash::uptr_t createHashPassword (HashFunctionCtx &hashCtx, const PasswordCacheCtx &password, bool isSession=false, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) noexcept(false)=0
- virtual RandomGeneratorCtx::sptr_t defaultRNG () noexcept(true)=0

- virtual void setDefaultRNG (RandomGeneratorCtx::sptr_t rng=nullptr) noexcept(false)=0
- virtual RandomGeneratorCtx::sptr_t createRandomGenerator (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual SymmetricBlockCipherCtx::uptr_t createSymmetricBlockCipher (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual SymmetricKeyWrapCtx::uptr_t createSymmetricKeyWrap (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual StreamCipherCtx::uptr_t createStreamCipher (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual AuthStreamCipherCtx::uptr_t createAuthStreamCipher (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual MessageAuthCodeCtx::uptr_t createMessageAuthCode (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual HashFunctionCtx::uptr_t createHashFunction (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual KeyDeriveFuncCtx::uptr_t createKeyDeriveFunc (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual KeyDiversifierCtx::uptr_t createKeyDiversifier (AlgId_t masterAlgId, std::size_t slaveKeyLength, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual EncryptPublicCtx::uptr_t createPublicEncryptor (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual DecryptPrivateCtx::uptr_t createPrivateDecryptor (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual KeyEncapsulatePublicCtx::uptr_t createPublicEncapsulator (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual KeyDecapsulatePrivateCtx::uptr_t createPrivateDecapsulator (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0

- virtual SigEncodePrivateCtx::uptr_t createPrivateSigEncoder (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual MsgRecoveryPublicCtx::uptr_t createPublicMsgRecovery (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual SignPrivateCtx::uptr_t createPrivateSigner (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual VerifyPublicCtx::uptr_t createPublicVerifier (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual KeyAgreePrivateCtx::uptr_t createPrivateKeyAgreement (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0
- virtual X509RequestSignerCtx::uptr_t createX509RequestSigner (AlgId_t algId, ReservedContextIndex_t reservedIndex=ALLOC_CONTEXT_ON_HEAP) noexcept(false)=0

Additional Inherited Members

9.27.1 Detailed Description

[SWS_CRYPTO_20700] Interface definition: ⌈ Crypto Provider is a "factory" interface of all supported Crypto Primitives and a "trusted environmet" for internal communications between them.

Note

All Crypto Primitives should have an actual reference to their parent Crypto Provider.
A Crypto Provider can be destroyed only after destroying of all its daughterly Crypto Primitives.
Each method of this interface that creates a Crypto Primitive instance is non-constant, because any such creation increases a references counter of the Crypto Primitive.
Any user of this interface should create shared pointers to it only by calls of the method shared_from_this()!

⌋*(RS_CRYPTO_02305, RS_CRYPTO_02307, RS_CRYPTO_02401)*

9.27.2 Member Typedef Documentation

9.27.2.1 typedef std::shared_ptr<CryptoProvider> sptr_t

[SWS_CRYPTO_20701] Interface definition: ⌈ Shared smart pointer of the interface.

⌋(*RS_CRYPTO_02311*)

9.27.2.2 using ObjectType = CryptoObject::Type

[SWS_CRYPTO_20702] Interface definition: ⌈ Enumeration of all types of crypto objects.

⌋(*RS_CRYPTO_02311*)

9.27.2.3 using AlgId_t = CryptoPrimitiveId::AlgId_t

[SWS_CRYPTO_20703] Interface definition: ⌈ A short alias for Algorithm ID type definition.

⌋(*RS_CRYPTO_02404*)

9.27.2.4 using ContainedContextsList_t = std::vector< std::pair<AlgId_t, bool> >

[SWS_CRYPTO_20704] Interface definition: ⌈ A list of Crypto Contexts that should be contained by a single shared memory slot in different time moments.

This vector indirectly specifies a minimal required capacity of a single reserved Context slot via a list of Contexts' IDs that must be hosted by the slot.

algId Specify target Crypto Primitive via it's algorithm ID.

bool Specify direct (`true`) or reverse (`false`) transformation.

Note

> This vector is used for calculation of minimal required capacity of the reserved Context slot.
> If at least one element of AlgId_t has value 0 (`ALGID_UNDEFINED`), then maximal supported Context size should be reserved.

⌋*(RS_CRYPTO_02404)*

9.27.2.5 using ContainedObjectsList_t = std::vector< std::pair<AlgId_t, ObjectType> >

[SWS_CRYPTO_20705] Interface definition: ⌈A list of Crypto Objects that should be contained by a single shared memory slot in different time moments.

This vector indirectly specifies a minimal required capacity of a single reserved Object slot via a list of Objects' IDs that must be hosted by the slot.

`AlgId_t` Specify target Crypto Primitive via it's algorithm ID.

`ObjectType` Specify concrete object type.

Note

> This vector is used for calculation of minimal required capacity of the reserved Object slot.
> If some `AlgId_t` element of the list has value 0 (`ALGID_UNDEFINED`), then maximal supported size of correspondent `ObjectType` should be reserved for this element.
> If at least one element `ObjectType` of the list has value 0 (`ObjectType::UNKNOWN`), then maximal supported object size should be reserved.

⌋*(RS_CRYPTO_02404)*

9.27.2.6 using ContextReservationMap_t = std::vector< ContainedContextsList_t >

[SWS_CRYPTO_20706] Interface definition: ⌈This vector specifies a whole mapping of minimally required capacities to the Context slots' indexes.

⌋*(RS_CRYPTO_02404)*

9.27.2.7 using ObjectReservationMap_t = std::vector< ContainedObjectsList_t >

[SWS_CRYPTO_20707] Interface definition: ⌈This vector specifies a whole mapping of minimally required capacities to the Object slots' indexes.

⌋(RS_CRYPTO_02404)

9.27.3 Constructor & Destructor Documentation

9.27.3.1 virtual ~CryptoProvider ( ) `[virtual]`, `[default]`

[SWS_CRYPTO_20710] Interface definition: ⌈ Destructor.

⌋(RS_CRYPTO_02311)

9.27.4 Member Function Documentation

9.27.4.1 virtual AlgId_t convertToAlgId (const std::string & *primitiveName*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20711] Interface definition: ⌈ Converts a common name of crypto algorithm to a correspondent vendor specific binary algorithm ID.

Parameters

`in`	*primitiveName*	A unified name of the crypto primitive (see "Crypto Primitives Naming Convention" for more details).

Returns

Vendor specific binary algorithm ID or `ALGID_UNDEFINED` if a primitive with provided name is not supported.

⌋(RS_CRYPTO_02308)

9.27.4.2 virtual std::string convertToAlgName (AlgId_t *algId*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20712] Interface definition: ⌈ Converts a vendor specific binary algorithm ID to a correspondent common name of the crypto algorithm.

Parameters

`in`	*algId*	A vendor specific binary algorithm ID.

Returns

A common name of the crypto algorithm (see "!_Naming_Convention.txt" for more details).

Exceptions

CryptoInvalidArgument	if `algId` argument has an unsupported value.

⌋(*RS_CRYPTO_02308*)

9.27.4.3 virtual void reserveContexts (const ContextReservationMap_t & *reservationMap*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20713] Interface definition: ⌈ Reserves memory for simultaneous hosting of all Contexts specified by the map.

Parameters

`in`	*reservationMap*	Centexts reservation map that defines minimal required size for each reserved Context slot.

Exceptions

CryptoInvalidArgument	if reservationMap includes unknown algorithm identifiers.
CryptoBadAlloc	if the requested reservation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.4 virtual void reserveObjects (const ObjectReservationMap_t & *reservationMap*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20714] Interface definition: ⌈ Reserves memory for simultaneous hosting of all Objects specified by the map.

Parameters

`in`	*reservationMap*	Objects reservation map that defines minimal required size for each reserved Object slot.

Exceptions

CryptoInvalidArgument	if unknown or unsupported combination of object type and algorithm ID presents in the reservationMap.
CryptoBadAlloc	if the requested reservation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.5 virtual void reserveContexts (std::size_t *quantity*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20715] Interface definition: ⌈ Reserves memory for simultaneous hosting of specified quantity of any type Contexts, i.e. maximal capacity will be reserved for each Context.

Parameters

`in`	*quantity*	Number of Centexts for reservation.

Exceptions

CryptoBadAlloc	if the requested reservation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.6 virtual void reserveObjects (std::size_t *quantity*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20716] Interface definition: ⌈ Reserves memory for simultaneous hosting of specified quantity of any type Objects, i.e. maximal capacity will be reserved for each Object.

Parameters

`in`	*quantity*	Number of Objects for reservation.

Exceptions

CryptoBadAlloc	if the requested reservation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.7 virtual void enterRealTimeMode ( ) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20717] Interface definition: ⌈ Switches the memory management emgine of the current execution thread to the "Real-Time" mode.

Note

After entering to the "Real-Time" mode, any allocations of Contexts or Objects on the heap are prohibited.
In the "Real-Time" mode only reserved Objects and Contexts can be used (see `reserveContexts()` and `reserveObjects()` methods and
Indexes of reserved Objects and Contexts slots are presented in the API by the types `ReservedObjectIndex_t` and `ReservedContextIndex_t` respectively.

⌋(*RS_CRYPTO_02406*)

9.27.4.8 virtual void leaveRealTimeMode ( ) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20718] Interface definition: ⌈ Switches the memory management emgine of the current execution thread to the "Non-Real-Time" mode.

Note

After leaving the "Real-Time" mode, any allocations of Contexts or Objects on the heap are allowed.
In the "Non-Real-Time" mode the reserved Objects and Contexts can be used too (see `reserveContexts()` and `reserveObjects()` methods).
Indexes of reserved Objects and Contexts slots are presented in the API by the types `ReservedObjectIndex_t` and `ReservedContextIndex_t` respectively.

⌋(*RS_CRYPTO_02406*)

9.27.4.9 virtual DomainParameters::sptr_t allocDomainParameters (AlgId_t *algId*, bool *isSession* = `false`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20719] Interface definition: ⌈ Allocates an empty object of domain parameters for specified algorithm.

Parameters

in	*algId*	An identifier of the algorithm for which the domain parameters are intended.
in	*isSession*	Defines the "session" (or "temporary") attribute for the target domain parameters (if `true`).
in	*isExportable*	Defines the exportability attribute for the target key context.
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Shared smart pointer to the allocated domain parameter object.

Exceptions

CryptoInvalidArgument	if `algId` has an incorrect value.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02113*, *RS_CRYPTO_02404*)

9.27.4.10 virtual DomainParameters::sptrc_t knownDomainParameters (const std::string & *oidName*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20720] Interface definition: ⌈ Loads a known domain parameters by their OID/Name.

Parameters

`in`	*oidName*	OID/Name of required domain parameters (names are case-insensitive).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Shared smart pointer to the allocated domain parameter object.

Note

Crypto Provider can share a single instance of named (i.e. constant) domain parameters between a few consumers.

Exceptions

CryptoInvalidArgument	if oidName has an incorrect value.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02404*)

9.27.4.11 virtual SymmetricKey::uptrc_t generateSymmetricKey (AlgId_t *algId*, Key::Usage_t *allowedUsage*, bool *isSession* = `true`, bool *isExportable* = `false`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20721] Interface definition: ⌈ Allocate a new symmetric key object and fill it by a new randomly generated value.

Parameters

`in`	*algId*	Identifier of target symmetric crypto algorithm.
`in`	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target key can be used (see constants in scope of `KeyMaterial`).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key (if `true`).
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Smart unique pointer to the created symmetric key object.

Note

> If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
> If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the generated key object.
> Any serializable (i.e. savable/non-session or exportable) key must generate own COUID!
> By default Crypto Provider should use an internal instance of a best from all supported RNG (ideally TRNG).

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02003, RS_CRYPTO_02101, RS_CRYPTO_02102, RS_CRYPTO_02107, RS_CRYPTO_02108, RS_CRYPTO_02111, RS_CRYPTO_02113, RS_CRYPTO_02115, RS_CRYPTO_02404)*

9.27.4.12 virtual PrivateKey::uptrc_t generatePrivateKey (AlgId_t *algId*, Key::Usage_t *allowedUsage*, bool *isSession* = `false`, bool *isExportable* = `false`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20722] Interface definition: ⌈ Allocate a new private key context of correspondent type and generates the key value randomly.

Parameters

`in`	*algId*	Identifier of target public-private key crypto algorithm.
`in`	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target key can be used (see constants in scope of `KeyMaterial`).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key (if `true`).

`in`	*params*	An optional pointer to Domain Parameters required for full specification of the transformation.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Smart unique pointer to the created private key object.

Note

A common COUID should be shared for both private and public keys.
Any serializable (i.e. savable/non-session or exportable) key must generate own COUID!
If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02003, RS_CRYPTO_02101, RS_CRYPTO_02102, RS_CRYPTO_02107, RS_CRYPTO_02108, RS_CRYPTO_02111, RS_CRYPTO_02113, RS_CRYPTO_02115, RS_CRYPTO_02404)*

9.27.4.13 virtual SecretSeed::uptrc_t generateSeed (AlgId_t *algId*, SecretSeed::Usage_t *allowedUsage*, bool *isSession* = `false`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20723] Interface definition: ⌈ Generate a random Secret Seed object of requested algorithm.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.

in	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target seed can be used (see constants in scope of `KeyMaterial`).
in	*isSession*	Defines the "session" (or "temporary") attribute for the target seed (if `true`).
in	*isExportable*	Defines the exportability attribute for the target seed (if `true`).
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to generated `SecretSeed` object.

Exceptions

InvalidArgument	if `seedSize` exceeds capacity of the reserved slot (defined by `reservedIndex`) or an implementation specific limitation.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

9.27.4.14 virtual std::size_t getSerializedSize (ObjectType *objectType*, AlgId_t *algId*, Serializable::FormatId_t *formatId* = Serializable::FORMAT_DEFAULT) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20724] Interface definition: ⌈ Returns required buffer size for serialization of an object in specific format.

Parameters

in	*objectType*	Type of the target object.
in	*algId*	A Crypto Provider algorithm ID of the target object.
in	*formatId*	A Crypto Provider specific identifier of the output format.

Returns

Size required for storing of the object serialized in the specified format.

Exceptions

CryptoRuntimeError	if the specified format ID is not supported for specified object type and algorithm ID.
CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.

⌋(*RS_CRYPTO_02404*)

9.27.4.15 virtual std::size_t getStorageSize (ObjectType *objectType*, AlgId_t *algId*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20725] Interface definition: ⌈ Returns required capacity of a key slot for saving of the object.

Parameters

`in`	*objectType*	Type of the target object.
`in`	*algId*	A Crypto Provider algorithm ID of the target object.

Returns

Size required for storing of the object in the Key Storage.

Exceptions

CryptoRuntimeError	if the specified object type and algorithm ID are incompatible with each other.
CryptoInvalidArgument	if any of arguments has incorrect value.

⌋(*RS_CRYPTO_02404*)

9.27.4.16 virtual TrustedContainer::uptr_t allocVolatileContainer (std::size_t *capacity* = *0*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20726] Interface definition: ⌈ Allocates a Volatile (virtual) Trusted Container according to directly specified capacity.

Parameters

`in`	*capacity*	Capacity required for this volatile trusted container (in bytes).

Returns

Unique smart pointer to an allocated volatile trusted container.

Note

The Volatile Trusted Container can be used for execution of the import operations.
Current process obtains the "Owner" rights for allocated Container.
If `(capacity == 0)` then the capacity of the container will be selected automatically according to a maximal size of supported crypto objects.
A few volatile (temporary) containers can coexist at same time without any affecting each-other.

Exceptions

CryptoRuntimeError	if unsupported combination of object type and algorithm ID presents in the list.
CryptoBadAlloc	if the requested allocation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.17 virtual TrustedContainer::uptr_t allocVolatileContainer (const ContainedObjectsList_t & *objectsList*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20727] Interface definition: ⌈ Allocates a Volatile (virtual) Trusted Container according to indirect specification of a minimal required capacity for hosting of any listed object.

Parameters

`in`	*objectsList*	A list of objects that can be stored to this volatile trusted container.

Returns

Unique smart pointer to an allocated volatile trusted container.

Note

The Volatile Trusted Container can be used for execution of the import operations.
Current process obtains the "Owner" rights for allocated Container.
Real container capacity is calculated as a maximal storage size of all listed objects.

Exceptions

CryptoBadAlloc	if the requested allocation cannot be executed.

⌋(*RS_CRYPTO_02404*)

9.27.4.18 virtual std::size_t exportSecuredObject (const CryptoObject & *object*, SymmetricKeyWrapCtx & *transportContext*, WritableMemRegion ∗ *serialized* = `nullptr`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20728] Interface definition: ⌈ Export a crypto object in a secure manner.

Parameters

`in`	*object*	A crypto object for export.
`in`	*transportContext*	A symmetric key wrap context initialized by a transport key (allowed usage: `ALLOW_KEY_EXPORTING`).
`out`	*serialized*	Optional pointer to output buffer for serialized data.

Returns

Actual capacity required for the serialized data.

Note

if `(serialized == nullptr)` then the method returns required size only, but content of the transportContext stays unchanged.
Only an exportable and completed object (i.e. that have a GUID) can be exported!

Exceptions

CryptoRuntimeError	if the transportContext is not initialized or its key doesn't have required attributes.
CryptoInvalidArgument	if size of the serialized is not enough for saving output data.

⌋(*RS_CRYPTO_02105*, *RS_CRYPTO_02112*)

9.27.4.19 virtual std::size_t exportSecuredObject (const TrustedContainer & *container,* SymmetricKeyWrapCtx & *transportContext,* WritableMemRegion ∗ *serialized* = `nullptr`) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20729] Interface definition: ⌈ Exports securely an object directly from a trusted container (i.e. without an intermediate creation of a crypto object).

Parameters

`in`	*container*	A trusted container that contains an object for export.
`in`	*transportContext*	A symmetric key wrap context initialized by a transport key (allowed usage: `ALLOW_KEY_EXPORTING`).
`out`	*serialized*	Optional pointer to output buffer for serialized data.

Returns

Actual capacity required for the serialized data.

Note

if `(serialized == nullptr)` then the method returns required size only, but content of the transportContext stays unchanged.
This method can be used for re-exporting of just imported object but on another transport key.

Exceptions

CryptoRuntimeError	if the `transportContext` is not initialized or its key doesn't have required attributes.
CryptoInvalidArgument	if size of the serialized is not enough for saving output data.

⌋*(RS_CRYPTO_02105, RS_CRYPTO_02112)*

9.27.4.20 virtual void importSecuredObject (TrustedContainer & *container,* ReadOnlyMemRegion *serialized,* SymmetricKeyWrapCtx & *transportContext,* bool *isExportable* = `false`, ObjectType *expectedObject* = ObjectType::UNKNOWN) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20730] Interface definition: ⌈ Imports securely serialized object to a temporary (volatile) trusted container for following processing (without allocation of a crypto object context).

Parameters

out	*container*	A prealocated volatile trusted container for storing of the imported object.
in	*serialized*	A memory region that contains a securely serialized object that should be imported to the trusted container.
in	*transportContext*	A symmetric key wrap context initialized by a transport key (allowed usage: `ALLOW_KEY_IMPORTING`).
in	*isExportable*	Defines the exportability attribute of the target object (this value for public keys and public domain parameters should be ignored).
in	*expectedObject*	Expected object type (default value ObjectType::UNKNOWN means without check).

Note

If `(expectedObject != ObjectType::UNKNOWN)` and an actual object type differs from the expected one then this method fails.
If the `transportContext` is not initialized or initialized by a key with incorrect attributes then this method fails.
If the `serialized` contains incorrect data then this method fails.

Exceptions

CryptoRuntimeError	if any of mentioned above fail conditions is satisfied.

⌋*(RS_CRYPTO_02105, RS_CRYPTO_02112, RS_CRYPTO_02113)*

9.27.4.21 virtual std::size_t exportPublicObject (const TrustedContainer & *container*, WritableMemRegion ∗ *serialized* = `nullptr`, Serializable::FormatId_t *formatId* = Serializable::FORMAT_DEFAULT) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20731] Interface definition: ⌈ Exports publicly an object from a trusted container (i.e. without an intermediate creation of a crypto object context).

Parameters

in	*container*	A trusted container that contains an object for export.
out	*serialized*	Optional pointer to output buffer for serialized data.
in	*formatId*	A Crypto Provider specific identifier of the output format.

Returns

Actual capacity required for the serialized data.

Exceptions

CryptoRuntimeError	if the container is empty.
CryptoInvalidArgument	if (serialized != nullptr), but its capacity is not enough for storing result.

⌋(*RS_CRYPTO_02105, RS_CRYPTO_02112*)

9.27.4.22 virtual void importPublicObject (TrustedContainer & *container*, ReadOnlyMemRegion & *serialized*, ObjectType *expectedObject* = ObjectType::UNKNOWN) [pure virtual], [noexcept]

[SWS_CRYPTO_20732] Interface definition: ⌈ Imports publicly serialized object to a temporary (volatile) trusted container for following processing (without allocation of a crypto object context).

Parameters

out	*container*	A prealocated volatile trusted container for storing of the imported object.
in	*serialized*	A memory region that contains a securely serialized object that should be imported to the trusted container.
in	*expectedObject*	Expected object type (default value ObjectType::UNKNOWN means without check).

Note

If (expectedObject != ObjectType::UNKNOWN) and an actual object type differs from the expected one then this method fails.
If the serialized contains incorrect data then this method fails.

Exceptions

CryptoRuntimeError	if any of mentioned above fail conditions is satisfied.

⌋(*RS_CRYPTO_02105, RS_CRYPTO_02112*)

9.27.4.23 virtual Key::uptrc_t loadKey (const TrustedContainer & *container*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20733] Interface definition: ⌈ Loads a whole key object (including its meta-data) to an opaque volatile key context of correspondent type.

Parameters

`in`	*container*	A trusted container that contains a key-object for loading.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created key object.

Exceptions

CryptoRuntimeError	if the container is empty or contains an object of type different from Key.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02404)*

9.27.4.24 virtual SecretSeed::uptrc_t loadSeed (const TrustedContainer & *container*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20734] Interface definition: ⌈ Loads a secret seed object (including its meta-data) to an opaque volatile seed context.

Parameters

`in`	*container*	A trusted container that contains a key-object for loading.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created secret seed object.

Exceptions

CryptoRuntimeError	if the container is empty or contains an object of type different from Seed.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02007, RS_CRYPTO_02404*)

9.27.4.25 virtual DomainParameters::uptrc_t loadDomainParameters (const TrustedContainer & *container,* ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20735] Interface definition: ⌈ Loads a domain parameters object (including its meta-data) to a volatile domain parameters context.

Parameters

`in`	*container*	A trusted container that contains a key-object for loading.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created secret seed object.

Exceptions

CryptoRuntimeError	if the container is empty or contains an object of type different from domain parameters.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02404*)

9.27.4.26 virtual PasswordCacheCtx::uptr_t allocPasswordCache (std::size_t *maximalLength*, std::size_t *requiredLength*, unsigned *requiredComplexity*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20736] Interface definition: ⌈ Allocates new Password Cache context.

Parameters

`in`	*maximalLength*	Maximal supported length of a target password (in characters).
`in`	*requiredLength*	Minimal required length of a target password (in characters).
`in`	*requiredComplexity*	Minimal required complexity of a target password (0 means "no requirements").
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Smart unique pointer to the allocated password context.

Note

The complexity is measured by a number symbols' categories (e.g.: lower/upper cases, numbers, special symbols).
A maximal supported value of the argument `maximalLength` can be restricted by an implementation.

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02106, RS_CRYPTO_02404*)

9.27.4.27 virtual PasswordHash::uptr_t loadPasswordHash (const TrustedContainer & *container*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20737] Interface definition: ⌈ Loads a password hash object to allocated context.

Parameters

in	*container*	A trusted container that contains the password hash object for loading.
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the allocated password hash context or nullptr if the container doesn't have an object of this type.

Note

An internal hash-function context required by the password hash context should be preallocated by this method too.

Exceptions

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02106, RS_CRYPTO_02404*)

9.27.4.28 virtual PasswordHash::uptr_t createHashPassword (HashFunctionCtx & *hashCtx*, const PasswordCacheCtx & *password*, bool *isSession* = `false`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20738] Interface definition: ⌈ Creates a password hash object.

Parameters

in	*hashCtx*	A hash-function context that should be used in this context.
in	*password*	An original password that should be hashed.
in	*isSession*	Defines the "session" (or "temporary") attribute for the password hash object (if `true`).
in	*isExportable*	Defines the exportability attribute for the password hash object (if `true`).
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the allocated password hash context or nullptr if the container doesn't have an object of this type.

Note

An internal hash-function context required by the password hash context should be preallocated by this method too.
Any serializable (i.e. savable/non-session or exportable) password hash object must generate own COUID!

Exceptions

CryptoRuntimeError	if configuration of the hash-function context (`hashCtx`) is not finished yet (i.e. Domain Parameters are required, but not provided yet).
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02106, RS_CRYPTO_02113, RS_CRYPTO_02404)*

9.27.4.29 virtual RandomGeneratorCtx::sptr_t defaultRNG ( ) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20739] Interface definition: ⌈ Returns a shared pointer to instance of default Random Number Generator (RNG) used by the Crypto Provider internaly.

Returns

Shared smart pointer to default Random Number Generator.

Note

The default RNG should be the best one (i.e. a most secure) from all supported RNG (ideally TRNG).

⌋*(RS_CRYPTO_02206, RS_CRYPTO_02404)*

9.27.4.30 virtual void setDefaultRNG (RandomGeneratorCtx::sptr_t *rng* = *nullptr*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20740] Interface definition: ⌈ Set default Random Number Generator (RNG) instance.

Parameters

`in`	*rng*	Shared smart pointer to an instance of the RNG context.

Note

if `(rng == nullptr)` then internal pointer of default RNG should be returned to original CryptoProider-specific default RNG instance.

Exceptions

CryptoRuntimeError	if `(rng != nullptr)`, but the provided RNG instance is not initialized yet.

⌋(*RS_CRYPTO_02206, RS_CRYPTO_02404*)

9.27.4.31 virtual RandomGeneratorCtx::sptr_t createRandomGenerator (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20741] Interface definition: ⌈ Creates a Random Number Generator (RNG) context.

Parameters

`in`	*algId*	Identifier of target RNG algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Shared smart pointer to the created RNG context.

Note

A fully Deterministic RNG should be used only for debugging purposes, but any RNG used "in the field" should support an internal entropy source (not controllable by application).

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02206, RS_CRYPTO_02404)*

9.27.4.32 virtual SymmetricBlockCipherCtx::uptr_t createSymmetricBlockCipher (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20742] Interface definition: ⌈ Creates a symmetric block cipher context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02404)*

9.27.4.33 virtual SymmetricKeyWrapCtx::uptr_t createSymmetricKeyWrap (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20743] Interface definition: ⌈ Creates a symmetric key-wrap context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from symmetric key-wrapping.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02104*, *RS_CRYPTO_02208*, *RS_CRYPTO_02404*)

9.27.4.34 virtual StreamCipherCtx::uptr_t createStreamCipher (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20744] Interface definition: ⌈ Creates a symmetric stream cipher context.

Parameters

in	*algId*	Identifier of target crypto algorithm.
in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from symmetric stream cipher.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02404)*

9.27.4.35 virtual AuthStreamCipherCtx::uptr_t createAuthStreamCipher (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20745] Interface definition: ⌈ Creates a symmetric authenticated stream cipher context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from symmetric authenticated stream cipher.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02207*, *RS_CRYPTO_02404*)

9.27.4.36 virtual MessageAuthCodeCtx::uptr_t createMessageAuthCode (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20746] Interface definition: ⌈ Creates a symmetric message authentication code context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from symmetric message authentication code.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02203*, *RS_CRYPTO_02404*)

9.27.4.37 virtual HashFunctionCtx::uptr_t createHashFunction (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20747] Interface definition: ⌈ Creates a hash function context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from hash function.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02205, RS_CRYPTO_02404*)

9.27.4.38 virtual KeyDeriveFuncCtx::uptr_t createKeyDeriveFunc (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20748] Interface definition: ⌈Creates a key derivation function context.

Parameters

`in`	*algId*	Identifier of target crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from key derivation function.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02103, RS_CRYPTO_02404*)

9.27.4.39 virtual KeyDiversifierCtx::uptr_t createKeyDiversifier (AlgId_t *masterAlgId,* std::size_t *slaveKeyLength,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20749] Interface definition: ⌈ Creates a symmetric key diversification context.

Parameters

`in`	*masterAlgId*	Crypto algorithm identifier of master-keys.
`in`	*slaveKeyLength*	Length of target slave-keys (derived from the master) in bits.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Note

slaveAlgId can have partial specification (only algorithm family and key length are required, but the mode and padding are optional).

Exceptions

CryptoInvalidArgument	if any of arguments has incorrect value or they are incompatible.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02103, RS_CRYPTO_02404)*

9.27.4.40 virtual EncryptPublicCtx::uptr_t createPublicEncryptor (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) [pure virtual], [noexcept]

[SWS_CRYPTO_20750] Interface definition: ⌈ Creates a public key encryption context.

Parameters

in	*algId*	Identifier of target asymmetric encryption/decryption algorithm.
in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric encryption/decryption.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02202, RS_CRYPTO_02404)*

9.27.4.41 virtual DecryptPrivateCtx::uptr_t createPrivateDecryptor (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20751] Interface definition: ⌈ Creates a private key decryption context.

Parameters

`in`	*algId*	Identifier of target asymmetric encryption/decryption algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric encryption/decryption.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02202, RS_CRYPTO_02404)*

9.27.4.42 virtual KeyEncapsulatePublicCtx::uptr_t createPublicEncapsulator (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20752] Interface definition: ⌈ Creates a public key context of a Key Encapsulation Mechanism (KEM).

Parameters

`in`	*algId*	Identifier of target KEM crypto algorithm.

in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric KEM.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02104, RS_CRYPTO_02209, RS_CRYPTO_02404)*

9.27.4.43 virtual KeyDecapsulatePrivateCtx::uptr_t createPrivateDecapsulator (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20753] Interface definition: ⌈Creates a private key context of a Key Encapsulation Mechanism (KEM).

Parameters

in	*algId*	Identifier of target KEM crypto algorithm.
in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric KEM.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02104, RS_CRYPTO_02209, RS_CRYPTO_02404*)

9.27.4.44 virtual SigEncodePrivateCtx::uptr_t createPrivateSigEncoder (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20754] Interface definition: ⌈ Creates a signature encoding private key context.

Parameters

in	*algId*	Identifier of target asymmetric crypto algorithm.
in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric signature encoding with message recovery.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02202, RS_CRYPTO_02204, RS_CRYPTO_02404*)

9.27.4.45 virtual MsgRecoveryPublicCtx::uptr_t createPublicMsgRecovery (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20755] Interface definition: ⌈ Creates a message recovery public key context.

Parameters

`in`	*algId*	Identifier of target asymmetric crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from asymmetric signature encoding with message recovery.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02202, RS_CRYPTO_02204, RS_CRYPTO_02404)*

9.27.4.46 virtual SignPrivateCtx::uptr_t createPrivateSigner (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20756] Interface definition: ⌈ Creates a private key signature context.

Parameters

`in`	*algId*	Identifier of target signature crypto algorithm.

in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from private key signature.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02204*, *RS_CRYPTO_02404*)

9.27.4.47 virtual VerifyPublicCtx::uptr_t createPublicVerifier (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20757] Interface definition: ⌈ Creates a public key verification context.

Parameters

in	*algId*	Identifier of target signature crypto algorithm.
in	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from public key signature verification.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02204, RS_CRYPTO_02404)*

9.27.4.48 virtual KeyAgreePrivateCtx::uptr_t createPrivateKeyAgreement (AlgId_t *algId,* ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20758] Interface definition: ⌈Creates a private key-agreement context.

Parameters

`in`	*algId*	Identifier of target key-agreement crypto algorithm.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from key-agreement.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02104, RS_CRYPTO_02404)*

9.27.4.49 virtual X509RequestSignerCtx::uptr_t createX509RequestSigner (AlgId_t *algId*, ReservedContextIndex_t *reservedIndex* = ALLOC_CONTEXT_ON_HEAP) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20759] Interface definition: ⌈ Creates a symmetric key diversification context.

Parameters

`in`	*algId*	Identifier of target signature crypto algorithm, that should be used for hashing and signature of certification requests produced by it.
`in`	*reservedIndex*	An optional index of reserved Context slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the created context.

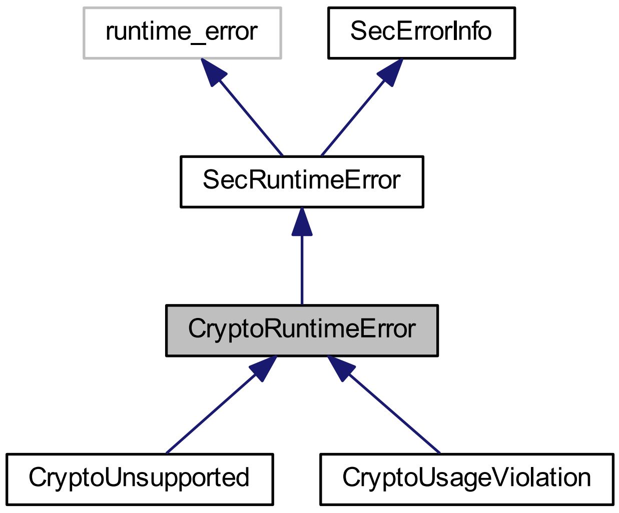
Exceptions

CryptoInvalidArgument	if `algId` specifies a crypto algorithm different from private key signature or doesn't include hash algorithm specification.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target context or if `(reservedIndex == ALLOC_CONTEXT_ON_HEAP)`, but allocation memory on the heap is impossible.

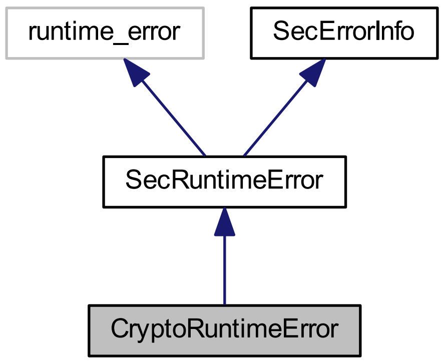
⌋*(RS_CRYPTO_02306, RS_CRYPTO_02307, RS_CRYPTO_02404)*

9.28 CryptoRuntimeError Interface Reference

Inheritance diagram for CryptoRuntimeError:



Collaboration diagram for CryptoRuntimeError:



Additional Inherited Members

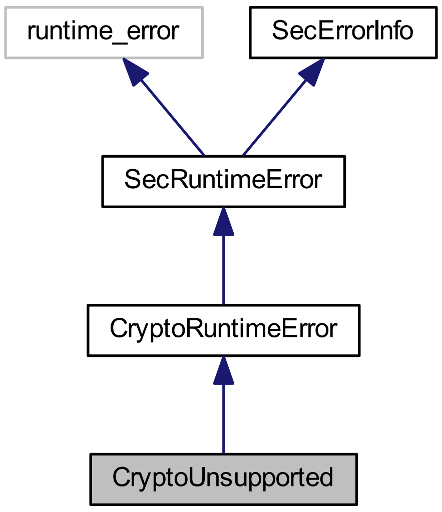
9.28.1 Detailed Description

[SWS_CRYPTO_29700] Interface definition: ⌈ A common interface for all Crypto Runtime Error exceptions

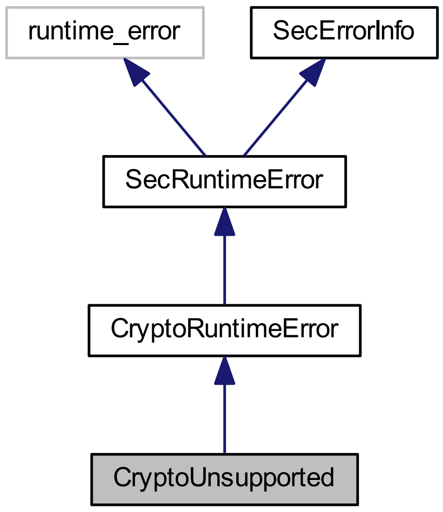
⌋*(RS_CRYPTO_02310)*

9.29 CryptoUnsupported Interface Reference

Inheritance diagram for CryptoUnsupported:



Collaboration diagram for CryptoUnsupported:



Additional Inherited Members

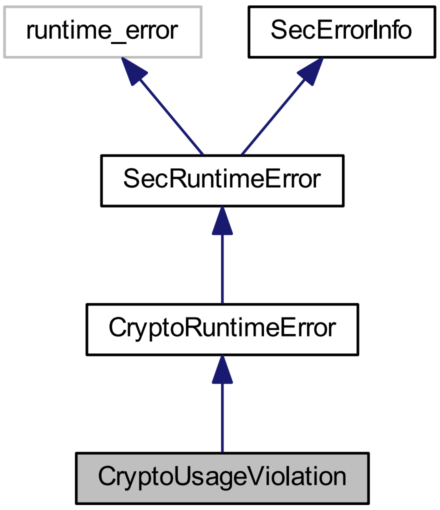
9.29.1 Detailed Description

[SWS_CRYPTO_29800] Interface definition: ⌈ An interface of the Crypto Unsupported method/argument exceptions

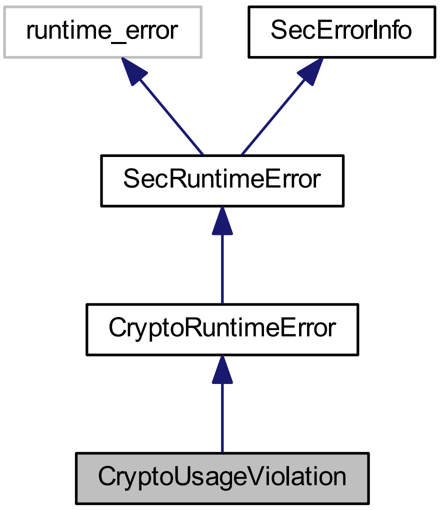
⌋*(RS_CRYPTO_02310)*

9.30 CryptoUsageViolation Interface Reference

Inheritance diagram for CryptoUsageViolation:



Collaboration diagram for CryptoUsageViolation:



Additional Inherited Members

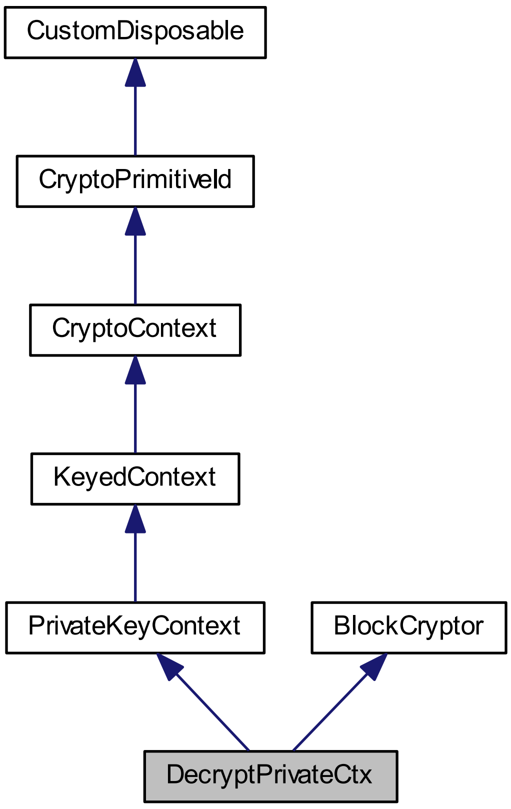
9.30.1 Detailed Description

[SWS_CRYPTO_29900] Interface definition: ⌈ An interface of the Cryptography Usage Violation exceptions

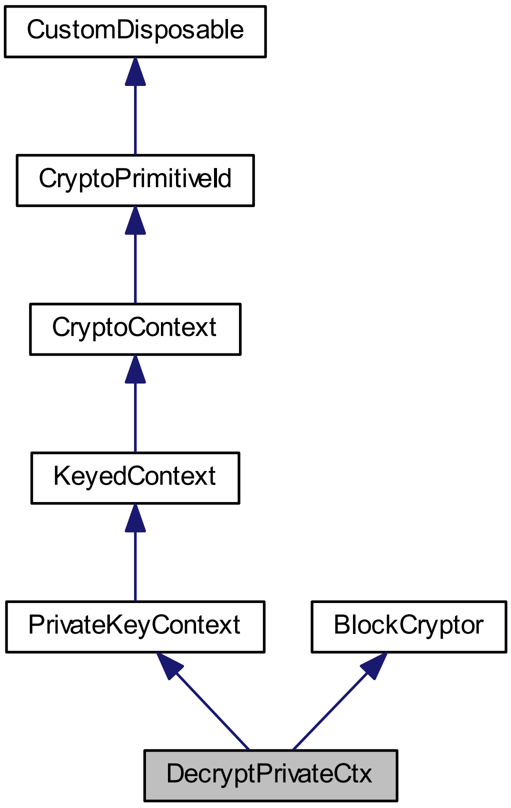
⌋ *(RS_CRYPTO_02310)*

9.31 DecryptPrivateCtx Interface Reference

Inheritance diagram for DecryptPrivateCtx:



Collaboration diagram for DecryptPrivateCtx:



Public Types

- typedef std::unique_ptr< DecryptPrivateCtx, CustomDeleter > uptr_t

Additional Inherited Members

9.31.1 Detailed Description

[SWS_CRYPTO_20800] Interface definition: ⌈ Asymmetric Decryption Private key Context interface

⌋*(RS_CRYPTO_02202)*

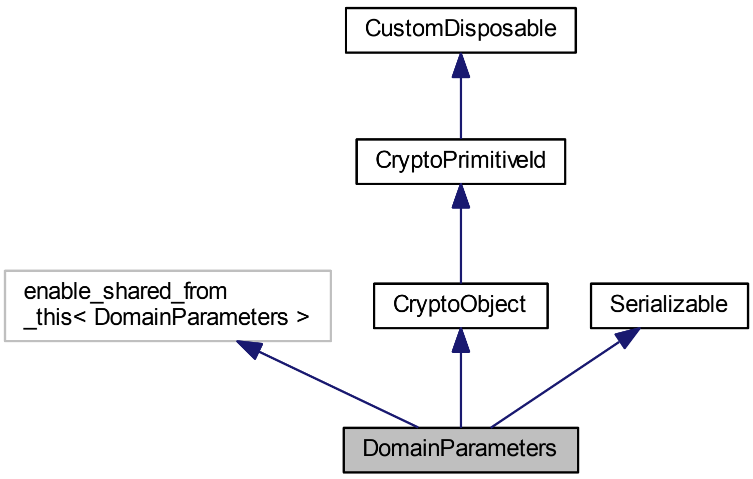
9.31.2 Member Typedef Documentation

9.31.2.1 typedef std::unique_ptr<DecryptPrivateCtx, CustomDeleter> uptr_t

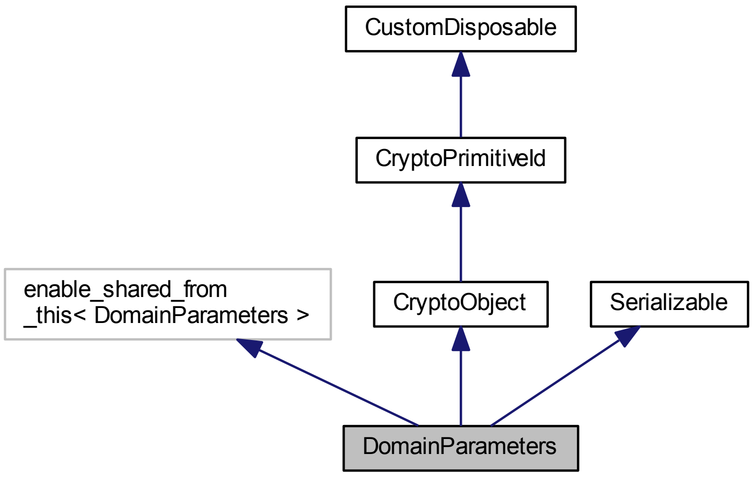
[SWS_CRYPTO_20801] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.32 DomainParameters Interface Reference

Inheritance diagram for DomainParameters:



Collaboration diagram for DomainParameters:



Public Types

- typedef std::shared_ptr$<$ DomainParameters $>$ sptr_t
- typedef std::shared_ptr$<$ const DomainParameters $>$ sptrc_t

Public Member Functions

- virtual bool isSecret () const noexcept(true)=0
- virtual std::size_t getParametersCount () const noexcept(true)=0
- virtual std::size_t getMaxParameterNameLength () const noexcept(true)=0
- virtual std::size_t getParameterName (std::size_t index, std::string ∗name=nullptr) const noexcept(false)=0
- virtual std::size_t expectedParameterSize (std::size_t index) const noexcept(false)=0
- virtual void setParameter (std::size_t index, ReadOnlyMemRegion value) noexcept(false)=0
- virtual bool complete () const noexcept(true)=0
- virtual bool complete () noexcept(true)=0
- virtual const std::string getUniqueName () const noexcept(true)=0

Additional Inherited Members

9.32.1 Detailed Description

[SWS_CRYPTO_20900] Interface definition: ⌈ Generic Domain Parameters interface

Note

> Any user of this interface should create shared pointers to it only by calls of the method `shared_from_this()`!

⌋*(RS_CRYPTO_02108)*

9.32.2 Member Typedef Documentation

9.32.2.1 typedef std::shared_ptr<DomainParameters> sptr_t

[SWS_CRYPTO_20901] Interface definition: ⌈ Shared smart pointer of the interface.

⌋*(RS_CRYPTO_02311)*

9.32.2.2 typedef std::shared_ptr<const DomainParameters> sptrc_t

[SWS_CRYPTO_20902] Interface definition: ⌈ Shared smart pointer of the constant interface.

⌋*(RS_CRYPTO_02311)*

9.32.3 Member Function Documentation

9.32.3.1 virtual bool isSecret ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20911] Interface definition: ⌈ Get secrecy attribute of these parameters set.

Returns

> true if the set of these parameters is secret.

⌋*(RS_CRYPTO_02309)*

9.32.3.2 virtual std::size_t getParametersCount ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20912] Interface definition: ⌈ Returns number of supported parameters.

Returns

Number of supported parameters.

⌋(*RS_CRYPTO_02309*)

9.32.3.3 virtual std::size_t getMaxParameterNameLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20913] Interface definition: ⌈ Returns maximal length between all names of this domain parameters.

Returns

Maximal length between all parameters' names.

⌋(*RS_CRYPTO_02309*)

9.32.3.4 virtual std::size_t getParameterName (std::size_t *index*, std::string ∗ *name* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20914] Interface definition: ⌈ Returns the parameter name by its index.

Parameters

`in`	*index*	Index of a parameter, name of which is requested. It should be less than `getParametersCount()`.
`out`	*name*	An optional pointer to a string where the parameter name should be saved.

Returns

Actual length of the parameter name.

Note

A list of supported parameters is specific for Crypto Primitive implementation and should be defined by a supplier.

Exceptions

CryptoInvalidArgument	if the index value is incorrect or if `(name != nullptr)`, but the `name->capacity()` is not enough to store the parameter name.

⌋(*RS_CRYPTO_02309*)

9.32.3.5 virtual std::size_t expectedParameterSize (std::size_t *index*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20915] Interface definition: ⌈ Gets expected size of specific parameter of the Crypto Primitive.

Parameters

`in`	*index*	Index of a parameter that should be set. It should be less than `getParametersCount()`.

Returns

Expected (presize or maximal) size of the parameter defined by the index.

Exceptions

CryptoInvalidArgument	if the index value is incorrect.

⌋*(RS_CRYPTO_02309)*

9.32.3.6 virtual void setParameter (std::size_t *index*, ReadOnlyMemRegion *value*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20916] Interface definition: ⌈ Sets a value to specific parameter of the Crypto Primitive.

Parameters

`in`	*index*	Index of a parameter, which should be set. It should be less than getParametersCount().
`in`	*value*	New value of the parameter that should be set.

Note

All Crypto Primitives that supports custom parameters also should have a correct default set of parameters.
A list of supported parameters is specific for Crypto Primitive implementation and should be defined by a supplier.
All named domain parameters (for which `getUniqueName()` returns non-empty string) are already created in the completed state and don't need a call of this method!

Exceptions

CryptoInvalidArgument	if the index or value are incorrect.

CryptoLogicError	if the domain parameters' set is already completed.

⌋(*RS_CRYPTO_02309*)

9.32.3.7 virtual bool complete ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_20917] Interface definition: ⌈ Checks completeness and consistency of this parameters set.

Returns

`true` if this set of domain parameters is completed and `false` otherwise.

Note

Until the set of domain parameters is incomplete the object COUID should not be set (i.e. equal to zero)!

⌋(*RS_CRYPTO_02311*)

9.32.3.8 virtual bool complete ( ) `[pure virtual], [noexcept]`

[SWS_CRYPTO_20918] Interface definition: ⌈ Checks completeness and consistency of this parameters set and generates the COUID if the parameters are completed at first time.

Returns

"true" if this set of domain parameters is completed and "false" otherwise.

Note

> Until the set of domain parameters is incomplete the object COUID should not be set (i.e. equal to zero)!
> If the set of domain parameters is already completed then all following calls of the method `setParameter()` will fail!
> All named domain parameters (for which `getUniqueName()` returns non-empty string) are already created in the completed state and have assigned COUID!
> Internal presence of a non-zero COUID is a marker of completed domain parameters, therefore the real check for completeness is required only if the COUID is zeros.

⌋*(RS_CRYPTO_02311)*

9.32.3.9 virtual const std::string getUniqueName ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_20919] Interface definition: ⌈ Returns a unique "well-known" name of the parameters set, if it is assigned (f.e. OID/Name).

Returns

> Unique name of the Domain Parameters set or the empty string if a unique name is not assigned.

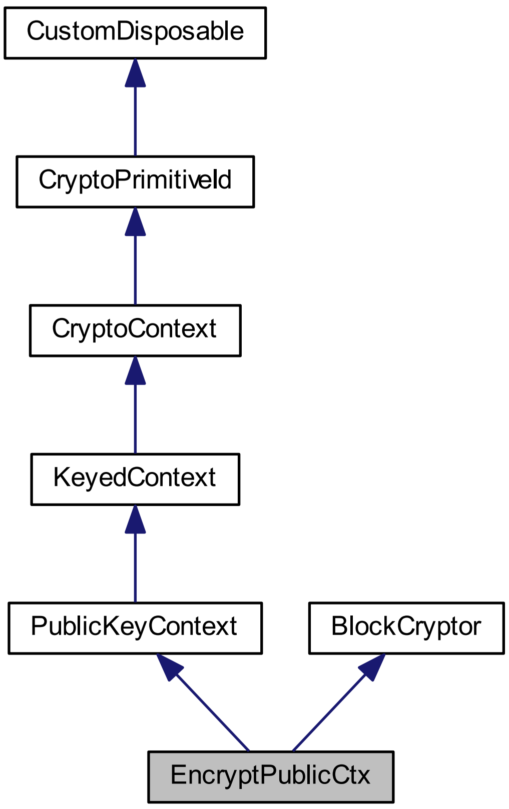
Note

> If this method returns non-empty string (i.e. all named domain parameters) then it is already in the completed state and has assigned COUID!

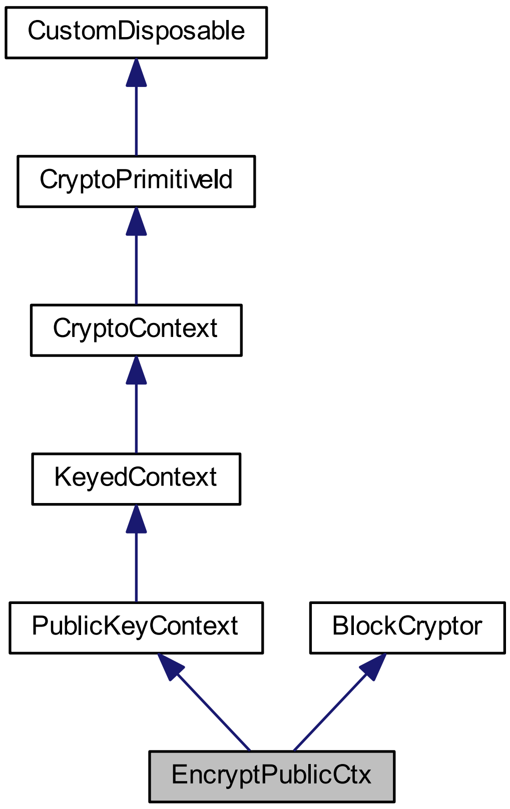
⌋*(RS_CRYPTO_02309)*

9.33 EncryptPublicCtx Interface Reference

Inheritance diagram for EncryptPublicCtx:



Collaboration diagram for EncryptPublicCtx:



Public Types

- typedef std::unique_ptr< EncryptPublicCtx, CustomDeleter > uptr_t

Additional Inherited Members

9.33.1 Detailed Description

[SWS_CRYPTO_21000] Interface definition: ⌈ Asymmetric Encryption Public key Context interface

⌋*(RS_CRYPTO_02202)*

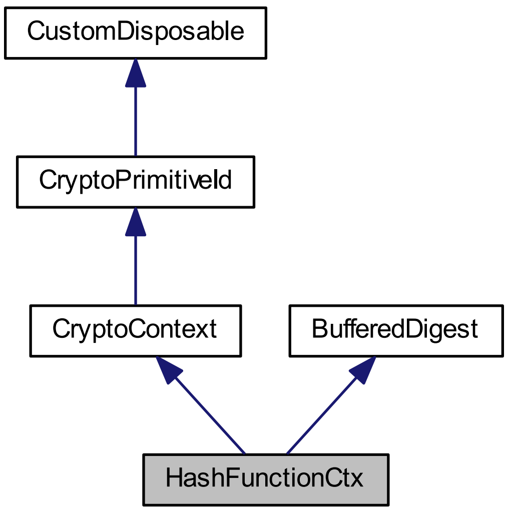
9.33.2 Member Typedef Documentation

9.33.2.1 typedef std::unique_ptr<EncryptPublicCtx, CustomDeleter> uptr_t

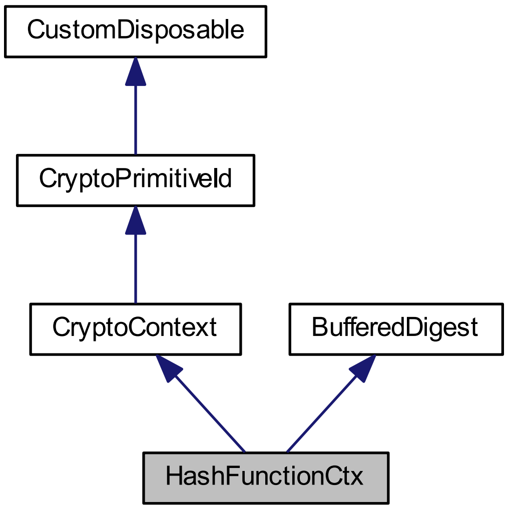
[SWS_CRYPTO_21001] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.34 HashFunctionCtx Interface Reference

Inheritance diagram for HashFunctionCtx:



Collaboration diagram for HashFunctionCtx:



Public Types

- typedef std::unique_ptr< HashFunctionCtx, CustomDeleter > uptr_t

Additional Inherited Members

9.34.1 Detailed Description

[SWS_CRYPTO_21100] Interface definition: ⌈ Hash function interface
⌋*(RS_CRYPTO_02205)*

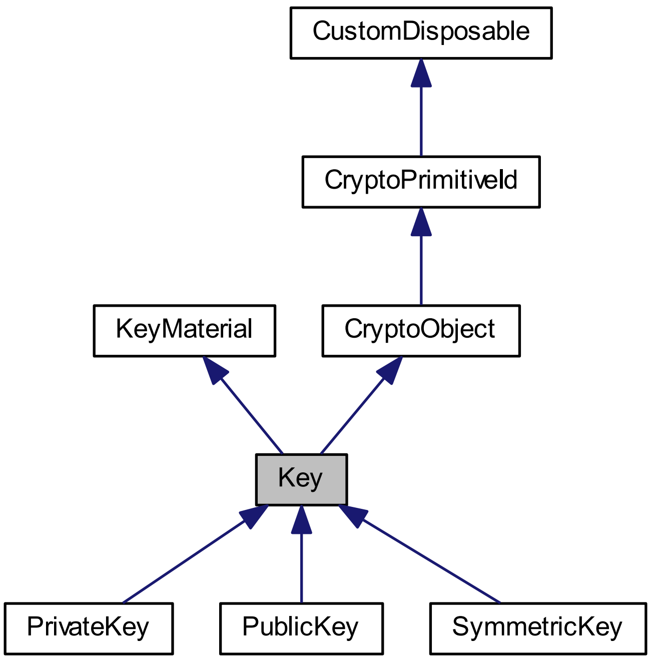
9.34.2 Member Typedef Documentation

9.34.2.1 typedef std::unique_ptr<HashFunctionCtx, CustomDeleter> uptr_t

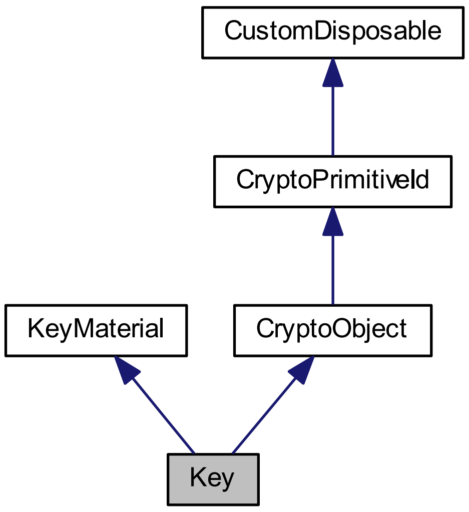
[SWS_CRYPTO_21101] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.35 Key Interface Reference

Inheritance diagram for Key:



Collaboration diagram for Key:



Public Types

- typedef std::unique_ptr< const Key, CustomDeleter > uptrc_t
- using Usage_t = AllowedUsageFlags_t

Public Member Functions

- virtual Usage_t allowedUsage () const noexcept(true)=0
- virtual bool isCompatible (AlgId_t algId) const noexcept(true)=0
- virtual bool isCompatible (const KeyedContext &context) const noexcept(true)=0
- virtual KeyType getKeyType () const noexcept(true)=0
- virtual bool isPublic () const noexcept(true)=0
- virtual bool isExportable () const noexcept(true)=0

Additional Inherited Members

9.35.1 Detailed Description

[SWS_CRYPTO_21200] Interface definition: ⌈ Generalized Key object interface

Note

> An implementation can filter allowed key values at generation/derivation time in order to prevent production of "weak" and "semi-weak" keys patterns specific to concrete algorithm.

⌋(*RS_CRYPTO_02403*)

9.35.2 Member Typedef Documentation

9.35.2.1 typedef std::unique_ptr<const Key, CustomDeleter> uptrc_t

[SWS_CRYPTO_21201] Interface definition: ⌈ Unique smart pointer of the interface.

⌋(*RS_CRYPTO_02404*)

9.35.2.2 using Usage_t = AllowedUsageFlags_t

[SWS_CRYPTO_21202] Interface definition: ⌈ Alias to the container type for bit-flags of allowed usages of the key.

⌋(*RS_CRYPTO_02008*)

9.35.3 Member Function Documentation

9.35.3.1 virtual Usage_t allowedUsage ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21211] Interface definition: ⌈ Returns allowed usages of the key.

Returns

> A combination of bit-flags that specifies allowed applications of the key.

⌋(*RS_CRYPTO_02008*)

9.35.3.2 virtual bool isCompatible (AlgId_t *algId*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21212] Interface definition: ⌈ Checks compatibility of this key with an algorithm specified by an ID.

Parameters

`in`	*algId*	Target Algorithm ID.

Returns

> `true` if the key is compatible with the algorithm specified by the algId and `false` otherwise.

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02309*)

9.35.3.3 virtual bool isCompatible (const KeyedContext & *context*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21213] Interface definition: ⌈ Checks compatibility of this key with a crypto transformation configured in the keyed context.

Parameters

`in`	*context*	Target keyed context.

Returns

> `true` if the key is compatible with the transformation of the context and `false` otherwise.

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02309*)

9.35.3.4 virtual KeyType getKeyType ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21214] Interface definition: ⌈ Returns actual type of this key object.

Returns

> Identifier of the top-level interface type of the key.

⌋(*RS_CRYPTO_02309*)

9.35.3.5 virtual bool isPublic ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21215] Interface definition: ⌈ Returns publicity of the key.

Returns

`true` if the key is public and false if the key is private or secret.

⌋*(RS_CRYPTO_02309)*

9.35.3.6 virtual bool isExportable ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21216] Interface definition: ⌈ Indicates a possibility to export the key.

Returns

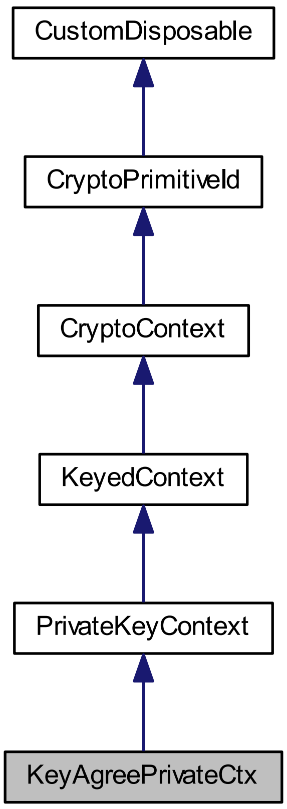
`true` if the key can be exported outside of the context (in any encrypted or plain format).

⌋*(RS_CRYPTO_02113, RS_CRYPTO_02309)*

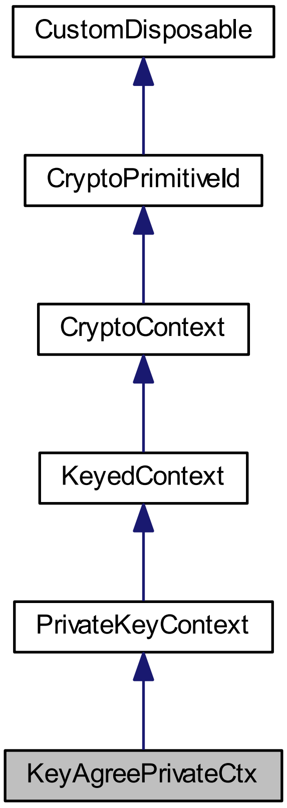
Implements CryptoObject.

9.36 KeyAgreePrivateCtx Interface Reference

Inheritance diagram for KeyAgreePrivateCtx:



Collaboration diagram for KeyAgreePrivateCtx:



Public Types

- typedef std::unique_ptr< KeyAgreePrivateCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual SecretSeed::uptrc_t agree (const PublicKey &otherSideKey, SecretSeed::Usage_t allowedUsage=ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual SymmetricKey::uptrc_t agree (const PublicKey &otherSideKey, KeyDeriveFuncCtx &kdf, AlgId_t targetAlgId, Key::Usage_t allowedUsage, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, Domain-

Parameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.36.1 Detailed Description

[SWS_CRYPTO_21300] Interface definition: ⌈ Key Agreement Private key Context interface.

⌋*(RS_CRYPTO_02104)*

9.36.2 Member Typedef Documentation

9.36.2.1 typedef std::unique_ptr<KeyAgreePrivateCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_21301] Interface definition: ⌈ Unique smart pointer of this interface.

⌋*(RS_CRYPTO_02404)*

9.36.3 Member Function Documentation

9.36.3.1 virtual SecretSeed::uptrc_t agree (const PublicKey & *otherSideKey,* SecretSeed::Usage_t *allowedUsage* = ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21311] Interface definition: ⌈ Execute the key-agreement algorithm between this private key and a public key of another side.

Parameters

`in`	*otherSideKey*	A public key of the other side of the Key-Agreement.
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to `SecretSeed` object, which contains the key material produced by the Key-Agreement algorithm.

Note

Produced `SecretSeed` object has following attributes: session, non-exportable, AlgID = this Key-Agreement AlgID.

Exceptions

CryptoLogicError	if the context was not initialized by a key value.
CryptoInvalidArgument	if the public and private keys correspond to different algorithms or reference to different domain parameters.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

9.36.3.2 virtual SymmetricKey::uptrc_t agree (const PublicKey & *otherSideKey,* KeyDeriveFuncCtx & *kdf,* AlgId_t *targetAlgId,* Key::Usage_t *allowedUsage,* ReadOnlyMemRegion ∗ *salt* `= nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* `= nullptr`, DomainParameters::sptrc_t *params* `= nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21312] Interface definition: ⌈Execute the key-agreement algorithm between this private key and a public key of another side.

Parameters

`in`	*otherSideKey*	A public key of the other side of the Key-Agreement.
`in`	*kdf*	Context of a Key Derivation Function, which should be used for the target key production.
`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target key-length).
`in`	*allowedUsage*	Allowed usage scope of the target key.
`in`	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).

`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to `SecretSeed` object, which contains the key material produced by the Key-Agreement algorithm.

Note

Produced `SymmetricKey` object has following attributes: session, non-exportable.
This method can be used for direct production of the target key, without creation of the intermediate `SecretSeed` object.

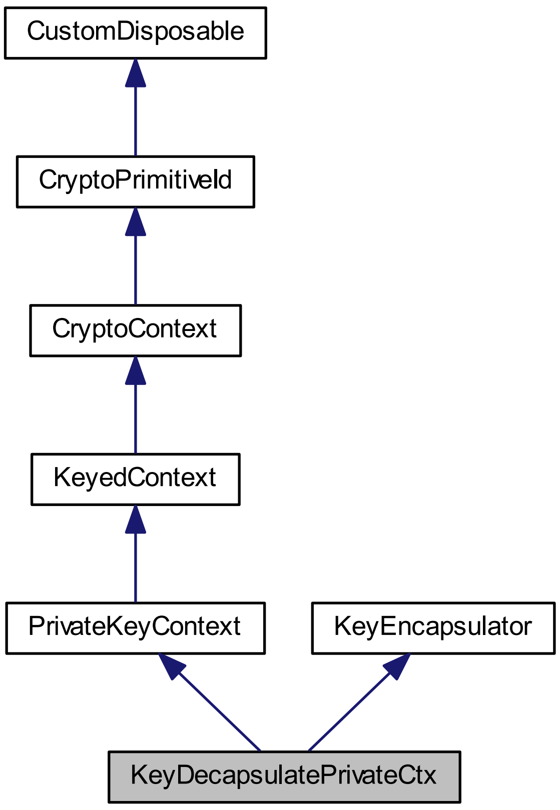
Exceptions

CryptoLogicError	if the context was not initialized by a key value.
CryptoInvalidArgument	if the public and private keys correspond to different algorithms or reference to different domain parameters.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

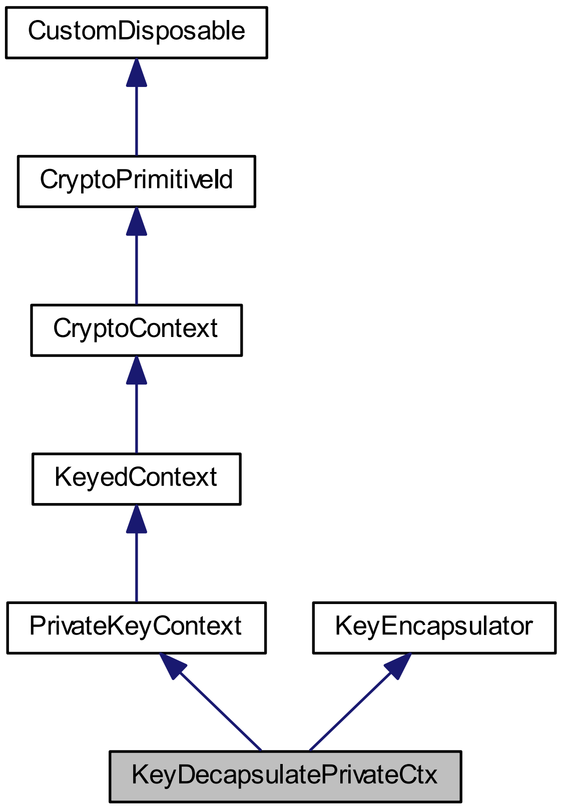
⌋*(RS_CRYPTO_02115, RS_CRYPTO_02404)*

9.37 KeyDecapsulatePrivateCtx Interface Reference

Inheritance diagram for KeyDecapsulatePrivateCtx:



Collaboration diagram for KeyDecapsulatePrivateCtx:



Public Types

- typedef std::unique_ptr< KeyDecapsulatePrivateCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual SecretSeed::uptrc_t decapsulateSeed (ReadOnlyMemRegion input, SecretSeed::Usage_t allowedUsage=ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual SymmetricKey::uptrc_t decapsulateKey (ReadOnlyMemRegion input, KeyDeriveFuncCtx &kdf, AlgId_t kekAlgId, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, DomainParame-

ters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.37.1 Detailed Description

[SWS_CRYPTO_21400] Interface definition: ⌈ Asymmetric Key Encapsulation Mechanism (KEM) Private key Context interface

⌋*(RS_CRYPTO_02104, RS_CRYPTO_02209, RS_CRYPTO_02404)*

9.37.2 Member Typedef Documentation

9.37.2.1 typedef std::unique_ptr<KeyDecapsulatePrivateCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_21401] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.37.3 Member Function Documentation

9.37.3.1 virtual SecretSeed::uptrc_t decapsulateSeed (ReadOnlyMemRegion *input*, SecretSeed::Usage_t *allowedUsage* = ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21411] Interface definition: ⌈ Decapsulates key material.

Parameters

`in`	*input*	An input buffer (its size should be equal `encapsulatedSize()` bytes).
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to `SecretSeed` object, which keeps the key material decapsulated from the input buffer.

Note

Returned Key Material object should be used for derivation of a symmetric key. Produced `SecretSeed` object has following attributes: session, non-exportable, AlgID = this KEM AlgID.

Exceptions

CryptoLogicError	if the context was not initialized by a public key value.
CryptoInvalidArgument	if the `output.size()` is not enough to store the result.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

9.37.3.2 virtual SymmetricKey::uptrc_t decapsulateKey (ReadOnlyMemRegion *input*, KeyDeriveFuncCtx & *kdf*, AlgId_t *kekAlgId*, ReadOnlyMemRegion ∗ *salt* = `nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21412] Interface definition: ⌈ Decapsulates Key Encryption Key (KEK).

Parameters

`in`	*input*	An input buffer (its size should be equal `encapsulatedSize()` bytes).
`in`	*kdf*	Context of a Key Derivation Function, which should be used for the target KEK production.
`in`	*kekAlgId*	Algorithm ID of target KEK.
`in`	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).

in	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
in	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to a Symmetric Key object derived from a key material decapsulated from the input block.

Note

Produced `SymmetricKey` object has following attributes: session, non-exportable, Key Usage: `ALLOW_KEY_IMPORTING`.
If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the generated key object.
This method can be used for direct production of the target key, without creation of the intermediate `SecretSeed` object.

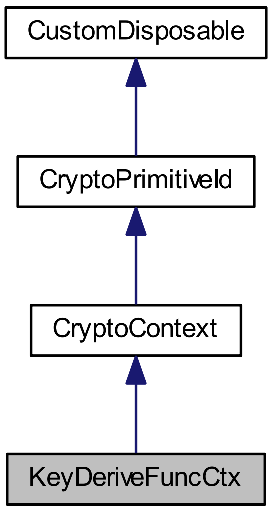
Exceptions

CryptoLogicError	if the context was not initialized by a private key value.
CryptoInvalidArgument	if `kekAlgId` specifies incorrect algorithm or the `input.size()` is not equal to `encapsulatedSize()`.
CryptoRuntimeError	if `(params != nullptr)`, but the domain parameters object has incomplete state.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

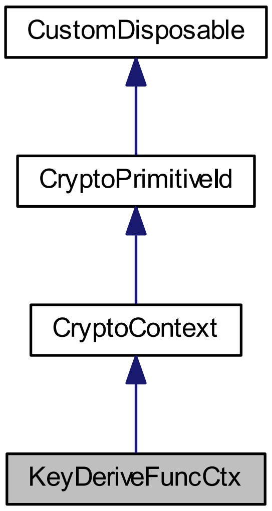
⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02108*, *RS_CRYPTO_02115*, *RS_CRYPTO_02404*)

9.38 KeyDeriveFuncCtx Interface Reference

Inheritance diagram for KeyDeriveFuncCtx:



Collaboration diagram for KeyDeriveFuncCtx:



Public Types

- typedef std::unique_ptr< KeyDeriveFuncCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::uint32_t configIterations (std::uint32_t iterations=0) noexcept(true)=0
- virtual SymmetricKey::uptrc_t deriveKey (AlgId_t targetAlgId, Key::Usage_t allowedUsage, const KeyMaterial &sourceKM, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, bool isSession=true, bool isExportable=false, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP, std::uint32_t iterations=0) const noexcept(false)=0
- virtual SymmetricKey::uptrc_t deriveKey (AlgId_t targetAlgId, Key::Usage_t allowedUsage, const KeyMaterial &sourceKM, const SecretSeed &salt, ReadOnlyMemRegion ∗ctxLabel=nullptr, bool isSession=true, bool isExportable=false, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP, std::uint32_t iterations=0) const noexcept(false)=0
- virtual SecretSeed::uptrc_t deriveSeed (AlgId_t targetAlgId, SecretSeed::Usage_t allowedUsage, const KeyMaterial &sourceKM, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, bool isSession=true, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP, std::uint32_t iterations=0) const noexcept(false)=0
- virtual SecretSeed::uptrc_t deriveSeed (AlgId_t targetAlgId, SecretSeed::Usage_t allowedUsage, const KeyMaterial &sourceKM, const SecretSeed &salt, ReadOnlyMemRegion ∗ctxLabel=nullptr, bool isSession=true, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP, std::uint32_t iterations=0) const noexcept(false)=0

Additional Inherited Members

9.38.1 Detailed Description

[SWS_CRYPTO_21500] Interface definition: ⌈ Key Derivation Function interface

⌋*(RS_CRYPTO_02103)*

9.38.2 Member Typedef Documentation

9.38.2.1 typedef std::unique_ptr<KeyDeriveFuncCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_21501] Interface definition: ⌈Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.38.3 Member Function Documentation

9.38.3.1 virtual std::uint32_t configIterations (std::uint32_t *iterations* = `0`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21511] Interface definition: ⌈Configure the number of iterations that will be applied by default.

Parameters

`in`	*iterations*	Requred number of the base function iterations (0 means implementation default number).

Returns

Actual number of iterations configured in the context now.

Note

Implementation can restrict minimal and/or maximal value of the iterations number.

⌋*(RS_CRYPTO_02309)*

9.38.3.2 virtual SymmetricKey::uptrc_t deriveKey (AlgId_t *targetAlgId*, Key::Usage_t *allowedUsage*, const KeyMaterial & *sourceKM*, ReadOnlyMemRegion ∗ *salt* = `nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, bool *isSession* = `true`, bool *isExportable* = `false`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP, std::uint32_t *iterations* = `0`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21512] Interface definition: ⌈Derive a symmetric key from provided key material (with optional public salt).

Parameters

`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target key-length).

`in`	*allowedUsage*	Allowed usage scope of the target key.
`in`	*sourceKM*	Source key material for the key derivation execution.
`in`	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key (if `true`).
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.
`in`	*iterations*	Requred number of base function iterations (0 means the current default number).

Returns

Unique smart pointer to created instance of derived symetric key.

Note

If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the produced key object.
The byte sequence provided via argument `ctxLabel` can include a few fields with different meaning separated by single `0x00` byte.

Exceptions

CryptoRuntimeError	if provided domain parameters object is not in the completed state or its content (at least algorithm) is incompatible with algId.
CryptoRuntimeError	if permissions associated with an object presented by the secret argument disallow its usage in this context.
CryptoInvalidArgument	if any of arguments is incorrect.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `sourceKM` object.

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02107*, *RS_CRYPTO_02108*, *RS_CRYPTO_02111*, *RS_CRYPTO_02113*, *RS_CRYPTO_02115*, *RS_CRYPTO_02404*)

9.38.3.3 virtual SymmetricKey::uptrc_t deriveKey (AlgId_t *targetAlgId*, Key::Usage_t *allowedUsage*, const KeyMaterial & *sourceKM*, const SecretSeed & *salt*, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, bool *isSession* = `true`, bool *isExportable* = `false`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP, std::uint32_t *iterations* = `0`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21513] Interface definition: ⌈ Derive a symmetric key from provided key material (with secret salt).

Parameters

`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target key-length).
`in`	*allowedUsage*	Allowed usage scope of the target key.
`in`	*sourceKM*	Source key material for the key derivation execution.
`in`	*salt*	A secret salt value (an additional secret value independent from the `sourceKM`).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key (if `true`).
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).

in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.
in	*iterations*	Requred number of base function iterations (0 means the current default number).

Returns

Unique smart pointer to created instance of derived symetric key.

Note

If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the produced key object.
The byte sequence provided via argument `ctxLabel` can include a few fields with different meaning separated by single `0x00` byte.

Exceptions

CryptoRuntimeError	if provided domain parameters object is not in the completed state or its content (at least algorithm) is incompatible with algId.
CryptoRuntimeError	if permissions associated with an object presented by the secret argument disallow its usage in this context.
CryptoInvalidArgument	if any of arguments is incorrect.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `sourceKM` or `salt` objects.

⌋*(RS_CRYPTO_02102, RS_CRYPTO_02107, RS_CRYPTO_02108, RS_CRYPTO_02111, RS_CRYPTO_02113, RS_CRYPTO_02115, RS_CRYPTO_02404)*

9.38.3.4 virtual SecretSeed::uptrc_t deriveSeed (AlgId_t *targetAlgId,* SecretSeed::Usage_t *allowedUsage,* const KeyMaterial & *sourceKM,* ReadOnlyMemRegion ∗ *salt* = `nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, bool *isSession* = `true`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP, std::uint32_t *iterations* = `0`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21514] Interface definition: ⌈ Derive a "slave" key material (secret seed) from provided "master" key material (with optional public salt).

Parameters

`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target seed-length).
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*sourceKM*	Source key material for the key derivation execution.
`in`	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key material (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key material (if `true`).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.
`in`	*iterations*	Requred number of base function iterations (0 means the current default number).

Returns

Unique smart pointer to created `SecretSeed` object, which keeps the derived key material.

Note

The byte sequence provided via argument `ctxLabel` can include a few fields with different meaning separated by single `0x00` byte.

Exceptions

CryptoRuntimeError	if permissions associated with an object presented by the secret argument disallow its usage in this context.
CryptoInvalidArgument	if any of arguments is incorrect.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `sourceKM` object.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02113, RS_CRYPTO_02404)*

9.38.3.5 virtual SecretSeed::uptrc_t deriveSeed (AlgId_t *targetAlgId*, SecretSeed::Usage_t *allowedUsage*, const KeyMaterial & *sourceKM*, const SecretSeed & *salt*, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, bool *isSession* = `true`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP, std::uint32_t *iterations* = *0*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21515] Interface definition: ⌈ Derive a "slave" key material (secret seed) from provided "master" key material (with secret salt).

Parameters

`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target seed-length).
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*sourceKM*	Source key material for the key derivation execution.
`in`	*salt*	A secret salt value (an additional secret value independent from the `sourceKM`).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key material (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key material (if `true`).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.
`in`	*iterations*	Requred number of base function iterations (0 means the current default number).

Returns

Unique smart pointer to created `SecretSeed` object, which keeps the derived key material.

Note

The byte sequence provided via argument `ctxLabel` can include a few fields with different meaning separated by single `0x00` byte.

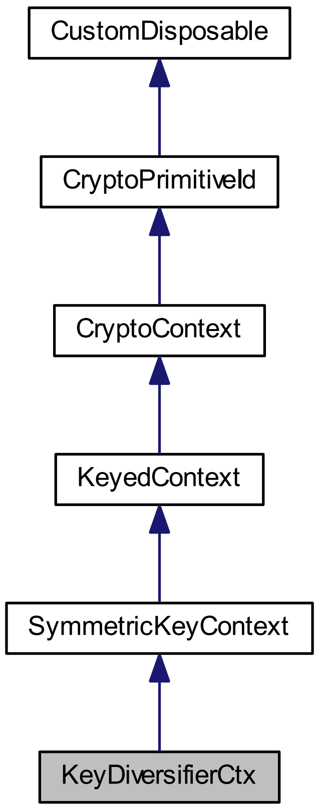
Exceptions

CryptoRuntimeError	if permissions associated with an object presented by the secret argument disallow its usage in this context.
CryptoInvalidArgument	if any of arguments is incorrect.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `sourceKM` or `salt` objects.

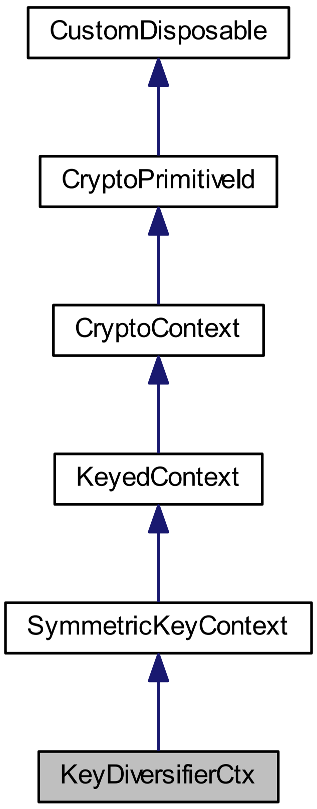
⌋*(RS_CRYPTO_02007, RS_CRYPTO_02113, RS_CRYPTO_02404)*

9.39 KeyDiversifierCtx Interface Reference

Inheritance diagram for KeyDiversifierCtx:



Collaboration diagram for KeyDiversifierCtx:



Public Types

- typedef std::unique_ptr< KeyDiversifierCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t getKeyIdSize () const noexcept(true)=0
- virtual std::size_t getFillerSize () const noexcept(true)=0
- virtual std::size_t getTargetKeyBitLength () const noexcept(true)=0
- virtual void init (ReadOnlyMemRegion appFiller, AlgId_t targetAlgId=ALGID_ANY, Key::Usage_t allowedUsage=ALLOW_PROTOTYPED_ONLY, DomainParameters::sptrc_t params=nullptr) noexcept(false)=0

- virtual void init (const SecretSeed &appFiller, AlgId_t targetAlgId=ALGID_ANY, Key::Usage_t allowedUsage=ALLOW_PROTOTYPED_ONLY, DomainParameters::sptrc_t params=nullptr) noexcept(false)=0
- virtual AlgId_t getTargetAlgId () const noexcept(true)=0
- virtual Key::Usage_t getTargetAllowedUsage () const noexcept(true)=0
- virtual SymmetricKey::uptrc_t diversify (ReadOnlyMemRegion targetKeyId, bool isSession=true, bool isExportable=false, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.39.1 Detailed Description

[SWS_CRYPTO_21600] Interface definition: ⌈ Interface of Symmetric Keys Diversification algorithms Context.

⌋*(RS_CRYPTO_02103)*

9.39.2 Member Typedef Documentation

9.39.2.1 typedef std::unique_ptr<KeyDiversifierCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_21601] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.39.3 Member Function Documentation

9.39.3.1 virtual std::size_t getKeyIdSize ( ) const [pure virtual], [noexcept]

[SWS_CRYPTO_21611] Interface definition: ⌈ Returns a fixed size of the target key ID required by diversification algorithm.

Returns

Size of the key ID in bytes.

Note

Returned value is constant for each instance of the interface, i.e. independent from configuration by the `init()` call.

⌋(*RS_CRYPTO_02311*)

9.39.3.2 virtual std::size_t getFillerSize ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21612] Interface definition: ⌈ Returns a fixed size of an application specific "filler" required by the diversificator initialization.

Returns

Size of the application specific filler in bytes.

Note

Returned value is constant for each instance of the interface, i.e. independent from configuration by the `init()` call.

⌋(*RS_CRYPTO_02311*)

9.39.3.3 virtual std::size_t getTargetKeyBitLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21613] Interface definition: ⌈ Returns the bit-length of target (diversified) keys.

Returns

The length of target (diversified) keys in bits.

Note

Returned value is configured by the context factory method, i.e. independent from configuration by the `init()` calls.

⌋(*RS_CRYPTO_02311*)

9.39.3.4 virtual void init (ReadOnlyMemRegion *appFiller*, AlgId_t *targetAlgId* = ALGID_ANY, Key::Usage_t *allowedUsage* = ALLOW_PROTOTYPED_ONLY, DomainParameters::sptrc_t *params* = *nullptr*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21614] Interface definition: ⌈ Executes initialisation of the diversifier context by setting a public "filler" value.

Parameters

in	*appFiller*	An application specific "filler" value.
in	*targetAlgId*	Identifier of target symmetric crypto algorithm.
in	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target key can be used.
in	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).

Note

If `(getFillerSize() == 0)` then the appFiller argument can have 0 size, because it is ignored in any case.
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the generated key object.
If `(targetAlgId == ALGID_ANY)` then a diversified key can be loaded to any symmetric context that supports same key length (if the "allowed usage" flags are also satisfied)!

Exceptions

CryptoInvalidArgument	if size of the `appFiller` is incorrect, i.e. if `(appFiller.size() < getFillerSize())`.
CryptoInvalidArgument	if `targetAlgId` specifies a cryptographic algorithm different from a symmetric one with key length equal to `getTargetKeyLength()`.
CryptoLogicError	if the context was not initialized by a key value.
CryptoRuntimeError	if `allowedUsage` specifies usage of the slave key incompatible with prototyped value of the master key loaded to the content.
CryptoRuntimeError	if `(params != nullptr)` and provided domain parameters object is in the incompleted state or has inappropriate type!

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02107*, *RS_CRYPTO_02108*, *RS_CRYPTO_02111*, *RS_CRYPTO_02115*)

9.39.3.5 virtual void init (const SecretSeed & *appFiller*, AlgId_t *targetAlgId* = ALGID_ANY, Key::Usage_t *allowedUsage* = ALLOW_PROTOTYPED_ONLY, DomainParameters::sptrc_t *params* = *nullptr*) [pure virtual], [noexcept]

[SWS_CRYPTO_21615] Interface definition: ⌈ Executes initialisation of the diversifier context by setting a secret "filler" value.

Parameters

in	*appFiller*	An application specific "filler" value.
in	*targetAlgId*	Identifier of target symmetric crypto algorithm.
in	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target key can be used.
in	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).

Note

If `(getFillerSize() == 0)` then the appFiller argument can have 0 size, because it is ignored in any case.
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the generated key object.
If `(targetAlgId == ALGID_ANY)` then a diversified key can be loaded to any symmetric context that supports same key length (if the "allowed usage" flags are also satisfied)!

Exceptions

CryptoInvalidArgument	if size of the appFiller is incorrect (i.e. `appFiller.size() < getFillerSize()`).
CryptoInvalidArgument	if `targetAlgId` specifies a cryptographic algorithm different from a symmetric one with key length equal to `getTargetKeyLength()`.
CryptoLogicError	if the context was not initialized by a key value.
CryptoRuntimeError	if `allowedUsage` specifies usage of the slave key incompatible with prototyped value of the master key loaded to the content.

CryptoRuntimeError	if `(params != nullptr)` and provided domain parameters object is in the incompleted state or has inappropriate type!
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `appFiller` object.

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02107*, *RS_CRYPTO_02108*, *RS_CRYPTO_02111*)

9.39.3.6 virtual AlgId_t getTargetAlgId ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21616] Interface definition: ⌈ Returns a symmetric algorithm ID of target (slave) keys.

Returns

Symmetric algorithm ID of target keys, configured by last call of the `init()` method.

Note

If the context was not configured yet by a call of the `init()` method then `ALGID_UNDEFINED` should be returned.

⌋(*RS_CRYPTO_02311*)

9.39.3.7 virtual Key::Usage_t getTargetAllowedUsage ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21617] Interface definition: ⌈ Returns allowed key usage of target (slave) keys.

Returns

Allowed key usage of target keys.

Note

Returned value depends from the master key prototype and the argument `targetAlgId` of last call of the `init()` method.
If the context was not configured yet by a call of the `init()` method then a prototyped value of the master key should be returned.

⌋(*RS_CRYPTO_02008*)

9.39.3.8 virtual SymmetricKey::uptrc_t diversify (ReadOnlyMemRegion *targetKeyId*, bool *isSession* = `true`, bool *isExportable* = `false`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21618] Interface definition: ⌈ Executes the key diversification from provided key ID.

Parameters

`in`	*targetKeyId*	ID of a target key.
`in`	*isSession*	Defines the "session" (or "temporary") attribute for the target key (if `true`).
`in`	*isExportable*	Defines the exportability attribute for the target key (if `true`).
`in`	*reservedIndex*	An optional index of reserved object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to the diversified slave key.

Exceptions

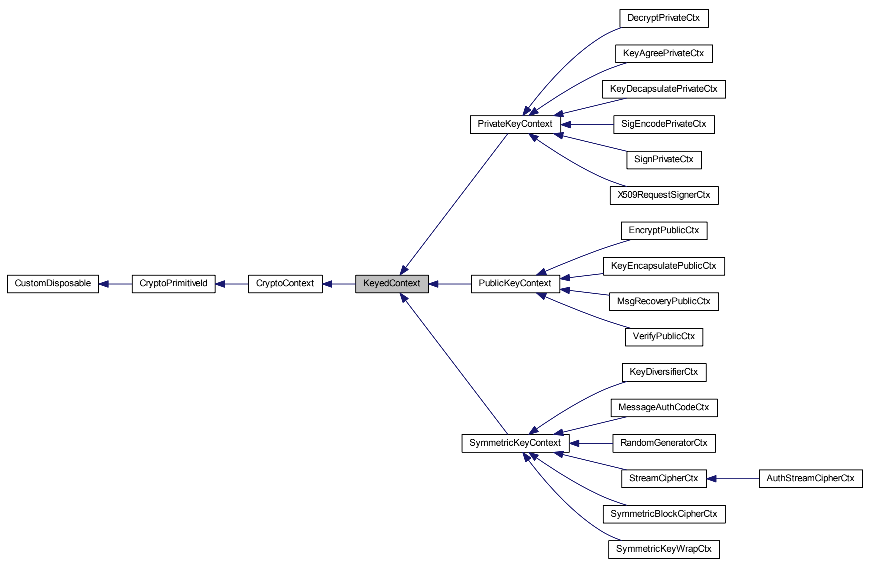
CryptoLogicError	if the context was not initialized by a key value.
CryptoLogicError	if `(0 < getFillerSize())`, but the context was not initialized by a "filler" value via call of the `init()` method.
CryptoInvalidArgument	if size of the `targetKeyId` are incorrect, e.g. if `(targetKeyId.size() < getKeyIdSize())`.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

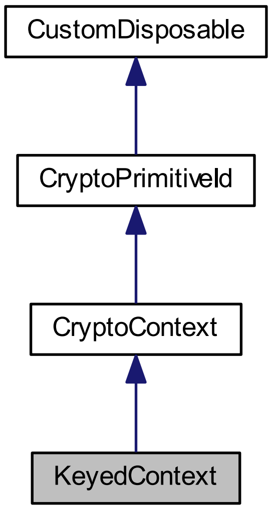
⌋(*RS_CRYPTO_02113*, *RS_CRYPTO_02404*)

9.40 KeyedContext Interface Reference

Inheritance diagram for KeyedContext:



Collaboration diagram for KeyedContext:



Public Member Functions

- virtual bool isKeyBitLengthSupported (std::size_t keyBitLength) const noexcept(true)=0
- virtual std::size_t minKeyBitLength () const noexcept(true)=0
- virtual std::size_t maxKeyBitLength () const noexcept(true)=0
- virtual std::size_t actualKeyBitLength (COUID ∗keyId=nullptr) const noexcept(true)=0

Additional Inherited Members

9.40.1 Detailed Description

[SWS_CRYPTO_21700] Interface definition: ⌈ A common interface of all keyed cryptographic contextes.

⌋*(RS_CRYPTO_02311)*

9.40.2 Member Function Documentation

9.40.2.1 virtual bool isKeyBitLengthSupported (std::size_t *keyBitLength*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21711] Interface definition: ⌈ Verify supportness of specific key length by the context.

Parameters

`in`	*keyBitLength*	Length of the key in bits.

Returns

true if provided value of the key length is supported by the context.

⌋*(RS_CRYPTO_02309)*

9.40.2.2 virtual std::size_t minKeyBitLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21712] Interface definition: ⌈ Returns minimal supported key length in bits.

Returns

Minimal supported length of the key in bits.

⌋*(RS_CRYPTO_02309)*

9.40.2.3 virtual std::size_t maxKeyBitLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21713] Interface definition: ⌈ Returns maximal supported key length in bits.

Returns

Maximal supported length of the key in bits.

⌋*(RS_CRYPTO_02309)*

9.40.2.4 virtual std::size_t actualKeyBitLength (COUID ∗ *keyId* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21714] Interface definition: ⌈ Returns actual bit-length of a key loaded to the context.

Parameters

`out`	*keyId*	Optional pointer to a buffer for saving an COUID of a key now loaded to the context.

Returns

Actual length of a key (now set to the algorithm context) in bits.

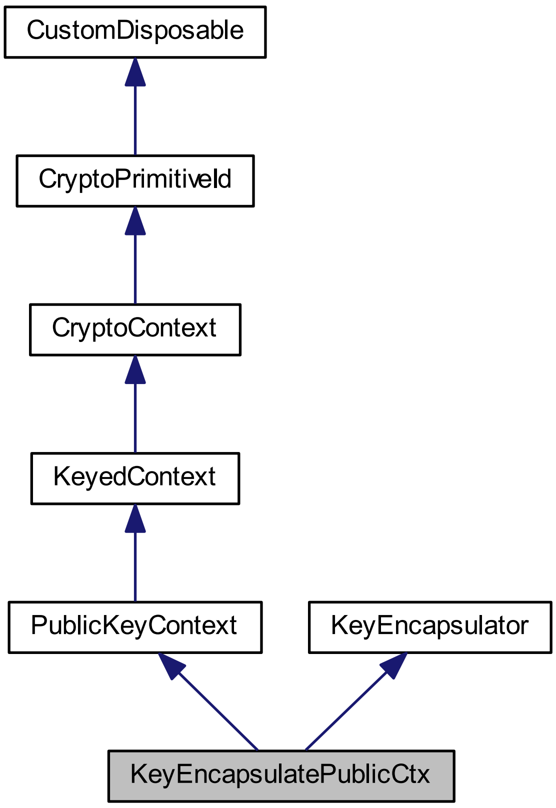
Note

If any key was not set to the context yet then 0 is returned.

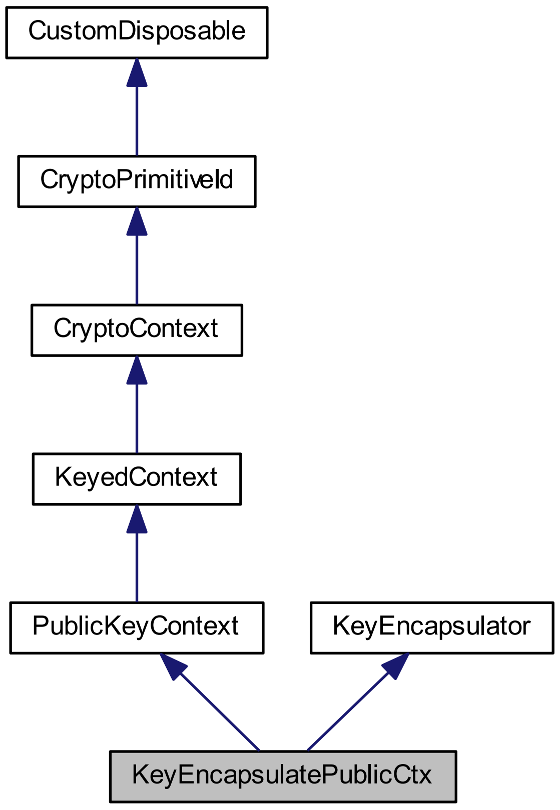
⌋(*RS_CRYPTO_02309*)

9.41 KeyEncapsulatePublicCtx Interface Reference

Inheritance diagram for KeyEncapsulatePublicCtx:



Collaboration diagram for KeyEncapsulatePublicCtx:



Public Types

- typedef std::unique_ptr< KeyEncapsulatePublicCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual SecretSeed::uptrc_t encapsulateSeed (WritableMemRegion output, std::size_t &outSize, SecretSeed::Usage_t allowedUsage=ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- template<typename Alloc >
SecretSeed::uptrc_t encapsulateSeed (std::vector< byte_t, Alloc > &output, SecretSeed::Usage_t allowedUsage=ALLOW_KDF_MATERIAL_ANY_USAGE,

ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)

- virtual SymmetricKey::uptrc_t encapsulateKey (WritableMemRegion output, std::size_t &outSize, KeyDeriveFuncCtx &kdf, AlgId_t kekAlgId, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- template<typename Alloc >
  SymmetricKey::uptrc_t encapsulateKey (std::vector< byte_t, Alloc > &output, KeyDeriveFuncCtx &kdf, AlgId_t kekAlgId, ReadOnlyMemRegion ∗salt=nullptr, ReadOnlyMemRegion ∗ctxLabel=nullptr, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)

Additional Inherited Members

9.41.1 Detailed Description

[SWS_CRYPTO_21800] Interface definition: ⌈ Asymmetric Key Encapsulation Mechanism (KEM) Public key Context interface

⌋*(RS_CRYPTO_02104, RS_CRYPTO_02209, RS_CRYPTO_02404)*

9.41.2 Member Typedef Documentation

9.41.2.1 typedef std::unique_ptr<KeyEncapsulatePublicCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_21801] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.41.3 Member Function Documentation

9.41.3.1 virtual SecretSeed::uptrc_t encapsulateSeed (WritableMemRegion *output*, std::size_t & *outSize*, SecretSeed::Usage_t *allowedUsage* = ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21811] Interface definition: ⌈ Encapsulates key material (seed).

Parameters

`out`	*output*	An output buffer (its size should be at least `outputSize()` bytes).
`out`	*outSize*	A variable for getting the actual size of output data (encapsulated key) in bytes.
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to `SecretSeed` obhect, which keeps the randomly renerated key material encapsulated to the output buffer.

Note

Returned key material should be used for derivation of a symmetric key.
Only first `encapsulatedSize()` bytes of the output buffer can be updated by this method.
Produced `SecretSeed` object has following attributes: session, non-exportable, AlgID = this KEM AlgID.

Exceptions

CryptoLogicError	if the context was not initialized by a public key value.
CryptoInvalidArgument	if the `output.size()` is not enough to store the result.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

9.41.3.2 SecretSeed::uptrc_t encapsulateSeed (std::vector< byte_t, Alloc > & *output*, SecretSeed::Usage_t *allowedUsage* = ALLOW_KDF_MATERIAL_ANY_USAGE, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_21812] Interface definition: ⌈ Encapsulates key encryption material.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*output*	A managed container for output data, i.e. for the "encapsulated key" (its capacity should be at least `outputSize()` bytes).
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to `SecretSeed` obhect, which keeps the randomly renerated key material encapsulated to the output buffer.

Note

Returned key material should be used for derivation of a symmetric key.
This method sets the size of the output container according to actually saved value.
Produced `SecretSeed` object has following attributes: session, non-exportable, AlgID = this KEM AlgID.

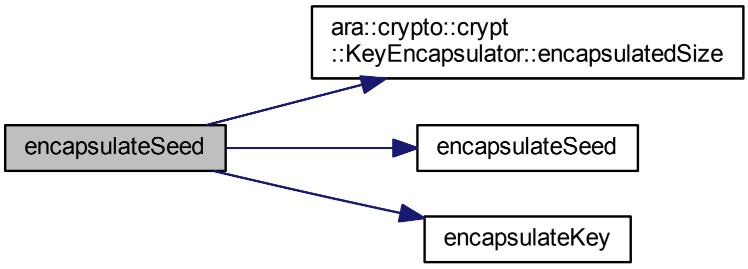
Exceptions

CryptoLogicError	if the context was not initialized by a public key value.
CryptoInvalidArgument	if the `output.capacity()` is not enough to store the result.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

Here is the call graph for this function:



9.41.3.3 virtual SymmetricKey::uptrc_t encapsulateKey (WritableMemRegion *output*, std::size_t & *outSize*, KeyDeriveFuncCtx & *kdf*, AlgId_t *kekAlgId*, ReadOnlyMemRegion ∗ *salt* = `nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_21813] Interface definition: ⌈ Encapsulates Key Encryption Key (KEK).

Parameters

`out`	*output*	An output buffer (its size should be at least `outputSize()` bytes).
`out`	*outSize*	A variable for getting the actual size of output data (encapsulated key) in bytes.
`in`	*kdf*	Context of a Key Derivation Function, which should be used for the target KEK production.
`in`	*kekAlgId*	Algorithm ID of target KEK.

in	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).
in	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
in	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to a Symmetric Key object derived from a randomly renerated material encapsulated to the output buffer.

Note

Only first `encapsulatedSize()` bytes of the output buffer should be updated by this method.
Produced `SymmetricKey` object has following attributes: session, non-exportable, Allowed Key Usage: `ALLOW_KEY_EXPORTING`.
If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the produced key object.
This method can be used for direct production of the target key, without creation of the intermediate `SecretSeed` object.

Exceptions

CryptoLogicError	if the context was not initialized by a public key value.
CryptoInvalidArgument	if `kekAlgId` specifies incorrect algorithm or the `output.size()` is not enough to store the result.
CryptoRuntimeError	if `(params != nullptr)`, but the domain parameters object has incomplete state.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02108*, *RS_CRYPTO_02115*, *RS_CRYPTO_02404*)

9.41.3.4 SymmetricKey::uptrc_t encapsulateKey (std::vector< byte_t, Alloc > & *output*, KeyDeriveFuncCtx & *kdf*, AlgId_t *kekAlgId*, ReadOnlyMemRegion ∗ *salt* = `nullptr`, ReadOnlyMemRegion ∗ *ctxLabel* = `nullptr`, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_21814] Interface definition: ⌈ Encapsulates Key Encryption Key (KEK).

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*output*	A managed container for output data, i.e. for the "encapsulated key" (its capacity should be at least `outputSize()` bytes).
`in`	*kdf*	Context of a Key Derivation Function, which should be used for the target KEK production.
`in`	*kekAlgId*	Algorithm ID of target KEK.
`in`	*salt*	Optional salt value (if used, it should be unique for each instance of the target key).
`in`	*ctxLabel*	Optional application specific "context label" (it can identify purpose of the target key and/or communication parties).
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to a Symmetric Key object derived from a randomly renerated material encapsulated to the output buffer.

Note

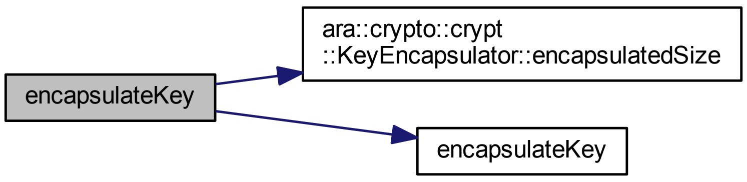
> Only first `encapsulatedSize()` bytes of the output buffer should be updated by this method.
> Produced `SymmetricKey` object has following attributes: session, non-exportable, Allowed Key Usage: `ALLOW_KEY_EXPORTING`.
> If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
> If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the produced key object.
> This method can be used for direct production of the target key, without creation of the intermediate `SecretSeed` object.

Exceptions

CryptoLogicError	if the context was not initialized by a public key value.
CryptoInvalidArgument	if `kekAlgId` specifies incorrect algorithm or the `output.size()` is not enough to store the result.
CryptoRuntimeError	if `(params != nullptr)`, but the domain parameters object has incomplete state.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

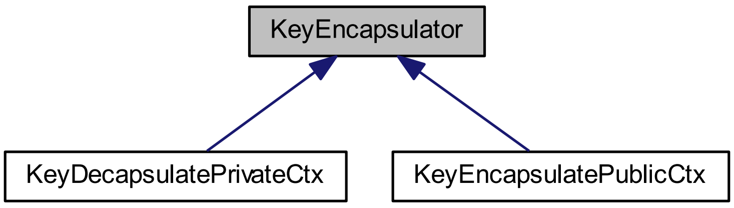
⌋(*RS_CRYPTO_02102*, *RS_CRYPTO_02108*, *RS_CRYPTO_02115*, *RS_CRYPTO_02404*)

Here is the call graph for this function:



9.42 KeyEncapsulator Interface Reference

Inheritance diagram for KeyEncapsulator:



Public Member Functions

- virtual std::size_t kekEntropy () const noexcept(true)=0
- virtual std::size_t encapsulatedSize () const noexcept(true)=0

Protected Member Functions

- virtual ∼KeyEncapsulator ()=default

9.42.1 Detailed Description

[SWS_CRYPTO_21900] Interface definition: ⌈ Basic interface of Asymmetric Key Encapsulation Mechanism (KEM)

⌋*(RS_CRYPTO_02209)*

9.42.2 Constructor & Destructor Documentation

9.42.2.1 virtual ∼KeyEncapsulator ( ) `[protected], [virtual], [default]`

[SWS_CRYPTO_21910] Interface definition: ⌈ Virtual destructor.

⌋*(RS_CRYPTO_02311)*

9.42.3 Member Function Documentation

9.42.3.1 virtual std::size_t kekEntropy ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21911] Interface definition: ⌈ Returns entropy (bit-length) of encapsulated key material.

Returns

Entropy of encapsulated key material in bits.

Note

For RSA system the returned value corresponds to the length of module N (minus 1).
For DH-like system the returned value corresponds to the length of module q (minus 1).

⌋*(RS_CRYPTO_02309)*

9.42.3.2 virtual std::size_t encapsulatedSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_21912] Interface definition: ⌈ Returns fixed size of the encapsulated data block.

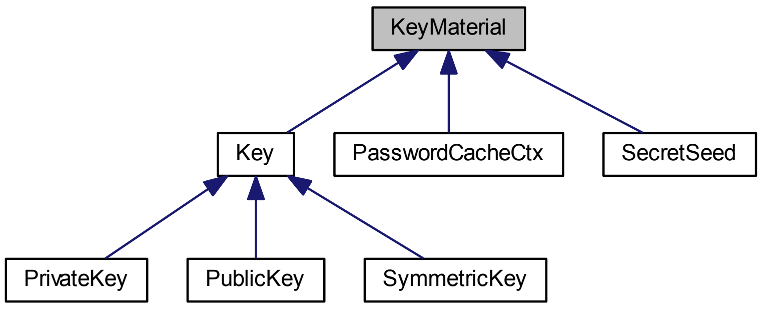
Returns

Size of the encapsulated data block in bytes.

⌋*(RS_CRYPTO_02309)*

9.43 KeyMaterial Interface Reference

Inheritance diagram for KeyMaterial:



Public Member Functions

- virtual std::size_t actualKeyBitLength () const noexcept(true)=0

9.43.1 Detailed Description

[SWS_CRYPTO_22000] Interface definition: ⌈ Generalized Key Material interface (in general case it represents a non-saveable temporary entity)

⌋*(RS_CRYPTO_02311)*

9.43.2 Member Function Documentation

9.43.2.1 virtual std::size_t actualKeyBitLength ( ) const [pure virtual], [noexcept]

[SWS_CRYPTO_22011] Interface definition: ⌈ Returns actual key length in bits.

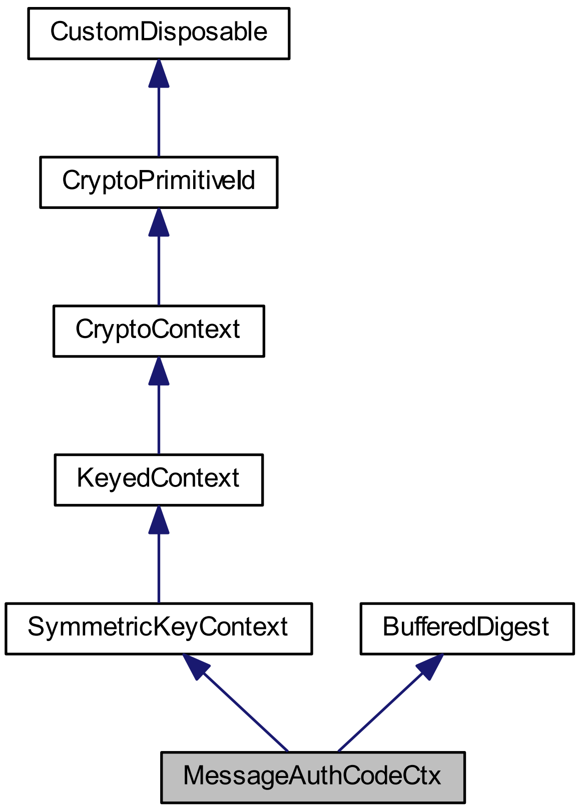
Returns

Actual key length in bits.

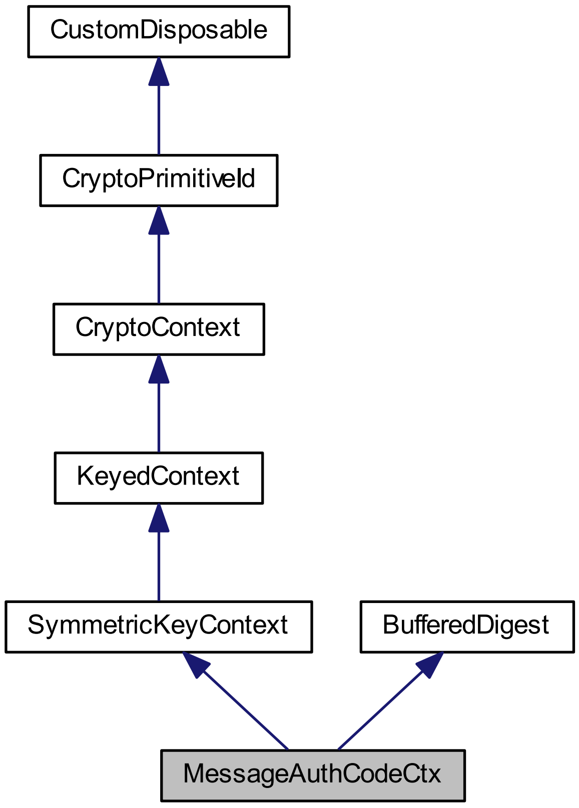
⌋(*RS_CRYPTO_02309*)

9.44 MessageAuthCodeCtx Interface Reference

Inheritance diagram for MessageAuthCodeCtx:



Collaboration diagram for MessageAuthCodeCtx:



Public Types

- typedef std::unique_ptr$<$ MessageAuthCodeCtx, CustomDeleter $>$ uptr_t

Additional Inherited Members

9.44.1 Detailed Description

[SWS_CRYPTO_22100] Interface definition: ⌈ Keyed Message Authentication Code Context interface definition (MAC/HMAC).

⌋*(RS_CRYPTO_02203)*

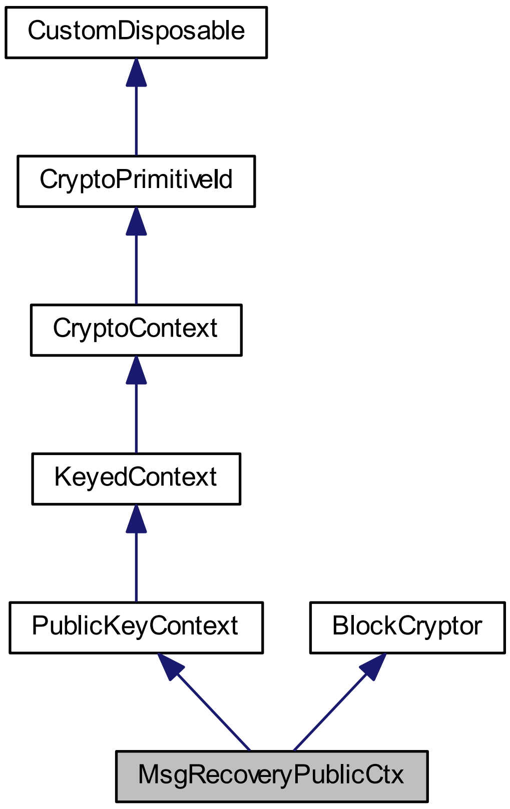
9.44.2 Member Typedef Documentation

9.44.2.1 typedef std::unique_ptr<MessageAuthCodeCtx, CustomDeleter> uptr_t

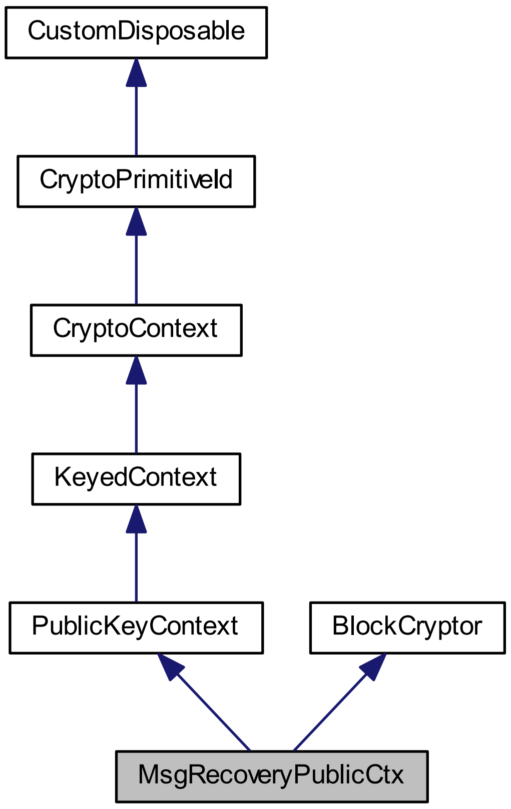
[SWS_CRYPTO_22101] Interface definition: ⌈ Unique smart pointer of the interface.
⌋ *(RS_CRYPTO_02404)*

9.45 MsgRecoveryPublicCtx Interface Reference

Inheritance diagram for MsgRecoveryPublicCtx:



Collaboration diagram for MsgRecoveryPublicCtx:



Public Types

- typedef std::unique_ptr< MsgRecoveryPublicCtx, CustomDeleter > uptr_t

Additional Inherited Members

9.45.1 Detailed Description

[SWS_CRYPTO_22200] Interface definition: ⌈ A public key context for asymmetric recovery of a short message and its signature verification (RSA-like).

Note

> Restricted groups of trusted subscribers can use this primitive for simultaneous provisioning of confidentiality, authenticity and non-repudiation of short messages, if the public key is generated appropriately and kept in secret.

If `(0 == BlockCryptor::processBlock(...))` then the input message-block is violated.

⌋(*RS_CRYPTO_02202*, *RS_CRYPTO_02204*)

9.45.2 Member Typedef Documentation

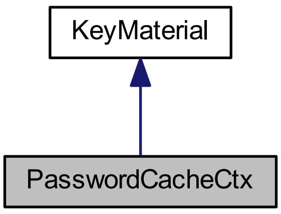
9.45.2.1 typedef std::unique_ptr<MsgRecoveryPublicCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_22201] Interface definition: ⌈ Unique smart pointer of the interface.

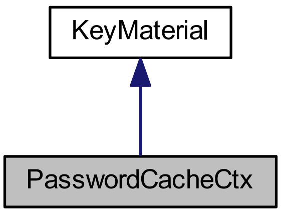
⌋(*RS_CRYPTO_02404*)

9.46 PasswordCacheCtx Interface Reference

Inheritance diagram for PasswordCacheCtx:



Collaboration diagram for PasswordCacheCtx:



Public Types

- typedef std::unique_ptr< PasswordCacheCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t maximalLength () const noexcept(true)=0
- virtual std::size_t requiredLength () const noexcept(true)=0
- virtual unsigned requiredComplexity () const noexcept(true)=0
- virtual void clear () noexcept(true)=0
- virtual std::size_t length () const noexcept(true)=0
- virtual unsigned complexity () const noexcept(true)=0
- virtual void reset (const std::string &password) noexcept(false)=0
- virtual bool compare (const std::string &password) const noexcept(true)=0
- virtual PasswordHash::uptr_t secureHash (HashFunctionCtx &hash, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual bool verify (HashFunctionCtx &hashCtx, const PasswordHash &passwordHash) const noexcept(false)=0

9.46.1 Detailed Description

[SWS_CRYPTO_22300] Interface definition: ⌈ Password secure cache context interface

⌋*(RS_CRYPTO_02106)*

9.46.2 Member Typedef Documentation

9.46.2.1 typedef std::unique_ptr<PasswordCacheCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_22301] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.46.3 Member Function Documentation

9.46.3.1 virtual std::size_t maximalLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22311] Interface definition: ⌈ Returns maximal password length.

Returns

Maximal supported password length (or buffer size) in characters.

⌋*(RS_CRYPTO_02311)*

9.46.3.2 virtual std::size_t requiredLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22312] Interface definition: ⌈ Returns required password length.

Returns

Minimal required password length in characters.

⌋*(RS_CRYPTO_02311)*

9.46.3.3 virtual unsigned requiredComplexity ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22313] Interface definition: ⌈ Returns required password complexity.

Returns

Minimal required password complexity (0 == no requirements).

Note

Each symbol category in requirements means +1 to the complexity (f.e.: lower/upper cases, numbers, special symbols).

⌋(*RS_CRYPTO_02311*)

9.46.3.4 virtual void clear ( ) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22314] Interface definition: ⌈ Securely clear the password cache.

⌋(*RS_CRYPTO_02311*)

9.46.3.5 virtual std::size_t length ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22315] Interface definition: ⌈ Returns actual password length.

Returns

Actual password length in characters.

⌋(*RS_CRYPTO_02311*)

9.46.3.6 virtual unsigned complexity ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22316] Interface definition: ⌈ Returns actual password complexity.

Returns

Actual password complexity level.

⌋(*RS_CRYPTO_02311*)

9.46.3.7 virtual void reset (const std::string & *password*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22317] Interface definition: ⌈ Reset password context to new value.

Parameters

`in`	*password*	A new value of the password.

Exceptions

CryptoRuntimeError	if the new password has a length greater than `maximalLength()` of this Password Cache instance.

⌋(*RS_CRYPTO_02311*)

9.46.3.8 virtual bool compare (const std::string & *password*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22318] Interface definition: ⌈ Compare password context with provided value for equality.

Parameters

`in`	*password*	An external value for comparison.

Returns

true if internally stored and provided passwords are equal.

⌋(*RS_CRYPTO_02311*)

9.46.3.9 virtual PasswordHash::uptr_t secureHash (HashFunctionCtx & *hash*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22319] Interface definition: ⌈ Calculate secure hash of the password randomized by a salt.

Parameters

`in`	*hash*	A hash-function context that should be used for the digest calculation.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to created Password Hash object.

Exceptions

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02404*)

9.46.3.10 virtual bool verify (HashFunctionCtx & *hashCtx*, const PasswordHash & *passwordHash*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22320] Interface definition: ⌈ Verifies a password stored in this context to compliance to the provided hash value.

Parameters

`in`	*hashCtx*	A hash context that should be used for verification.
`in`	*passwordHash*	A password hash object containing a reference value.

Returns

`true` if a password stored in this context matches to provided hash value.

Note

Before returning from this method the hashCtx context should be cleaned (from intermediate results)!

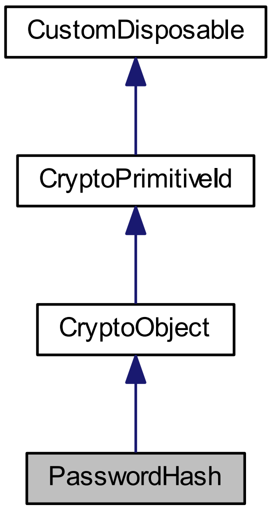
Exceptions

CryptoRuntimeError	if the hash algorithms of the hash-function context and Password Hash object differ.

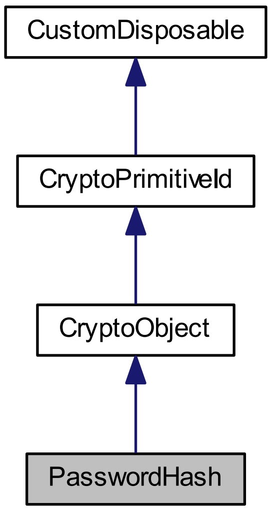
⌋(*RS_CRYPTO_02311*)

9.47 PasswordHash Interface Reference

Inheritance diagram for PasswordHash:



Collaboration diagram for PasswordHash:



Public Types

- typedef std::unique_ptr< PasswordHash, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t hashSize () const noexcept(true)=0
- virtual std::size_t seedSize () const noexcept(true)=0

Additional Inherited Members

9.47.1 Detailed Description

[SWS_CRYPTO_22400] Interface definition: ⌈ Secure Password Hash object interface.
Note

> The password hash object cannot be exportable!
> This object includes "seed" (randomization vector) and hash value of the password and seed.

⌋*(RS_CRYPTO_02106)*

9.47.2 Member Typedef Documentation

9.47.2.1 typedef std::unique_ptr<PasswordHash, CustomDeleter> uptr_t

[SWS_CRYPTO_22401] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.47.3 Member Function Documentation

9.47.3.1 virtual std::size_t hashSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_22411] Interface definition: ⌈ Returns a hash size of this object.

Returns

Size of the hash value in bytes.

⌋*(RS_CRYPTO_02309)*

9.47.3.2 virtual std::size_t seedSize ( ) const [pure virtual], [noexcept]

[SWS_CRYPTO_22412] Interface definition: ⌈ Returns a seed size of this object.

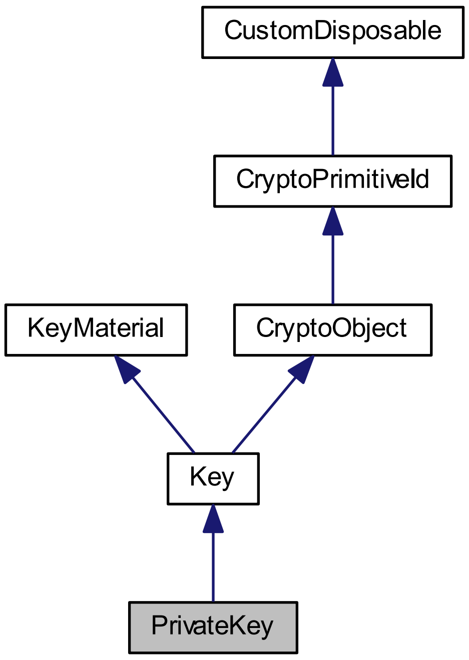
Returns

Size of the seed value in bytes.

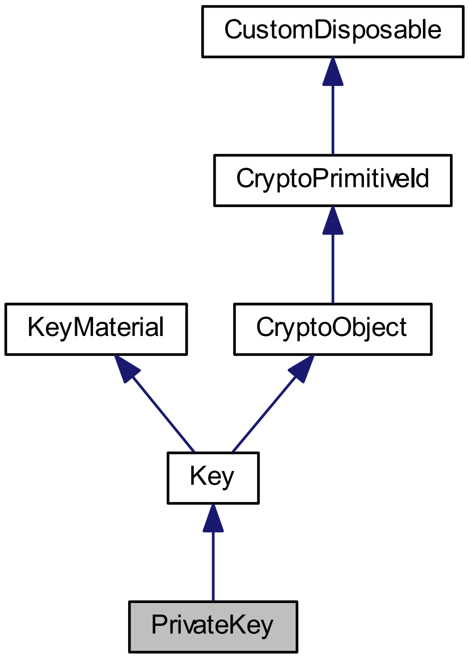
⌋*(RS_CRYPTO_02309)*

9.48 PrivateKey Interface Reference

Inheritance diagram for PrivateKey:



Collaboration diagram for PrivateKey:



Public Types

- typedef std::unique_ptr< const PrivateKey, CustomDeleter > uptrc_t

Public Member Functions

- virtual PublicKey::uptrc_t getPublicKey (DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.48.1 Detailed Description

[SWS_CRYPTO_22500] Interface definition: ⌈ Generalized Asymmetric Private Key interface

⌋*(RS_CRYPTO_02002, RS_CRYPTO_02403)*

9.48.2 Member Typedef Documentation

9.48.2.1 typedef std::unique_ptr<const PrivateKey, CustomDeleter> uptrc_t

[SWS_CRYPTO_22501] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.48.3 Member Function Documentation

9.48.3.1 virtual PublicKey::uptrc_t getPublicKey (DomainParameters::sptrc_t *params* = nullptr, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const [pure virtual], [noexcept]

[SWS_CRYPTO_22502] Interface definition: ⌈ Getter of the public key correspondent to this private key.

Parameters

in	*params*	An optional pointer to Domain Parameters required for full specification of the transformation.
in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A unique pointer to the public key correspondent to this private key.

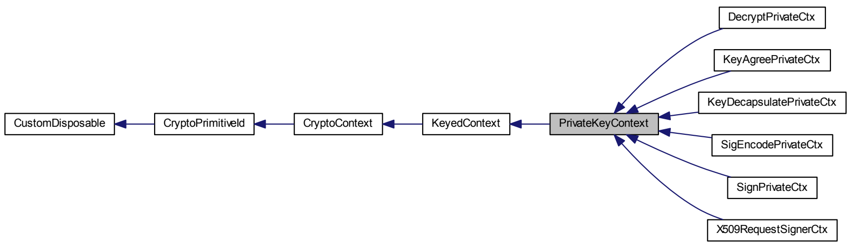
Exceptions

CryptoInvalidArgument	if provided domain parameters value is incorrect for this algorithm.
CryptoRuntimeError	if provided domain parameters object has `COUID` different from the `COUID` referenced in this private key (i.e. returned by `this->hasDependence()`).
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

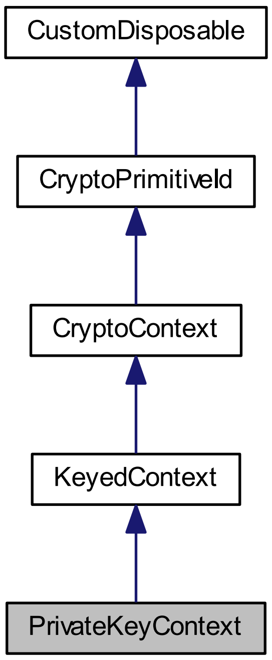
⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02115*, *RS_CRYPTO_02404*)

9.49 PrivateKeyContext Interface Reference

Inheritance diagram for PrivateKeyContext:



Collaboration diagram for PrivateKeyContext:



Public Member Functions

- virtual void setKey (const PrivateKey &key, DomainParameters::sptrc_t params=nullptr) noexcept(false)=0

Additional Inherited Members

9.49.1 Detailed Description

[SWS_CRYPTO_22600] Interface definition: ⌈ Generalized interface of Private Key algorithm Context

⌋*(RS_CRYPTO_02202)*

9.49.2 Member Function Documentation

9.49.2.1 virtual void setKey (const PrivateKey & *key*, DomainParameters::sptrc_t *params* = `nullptr`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22611] Interface definition: ⌈ Sets (deploy) a key to the algorithm context.

Parameters

`in`	*key*	A reference to a source key.
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the transformation.

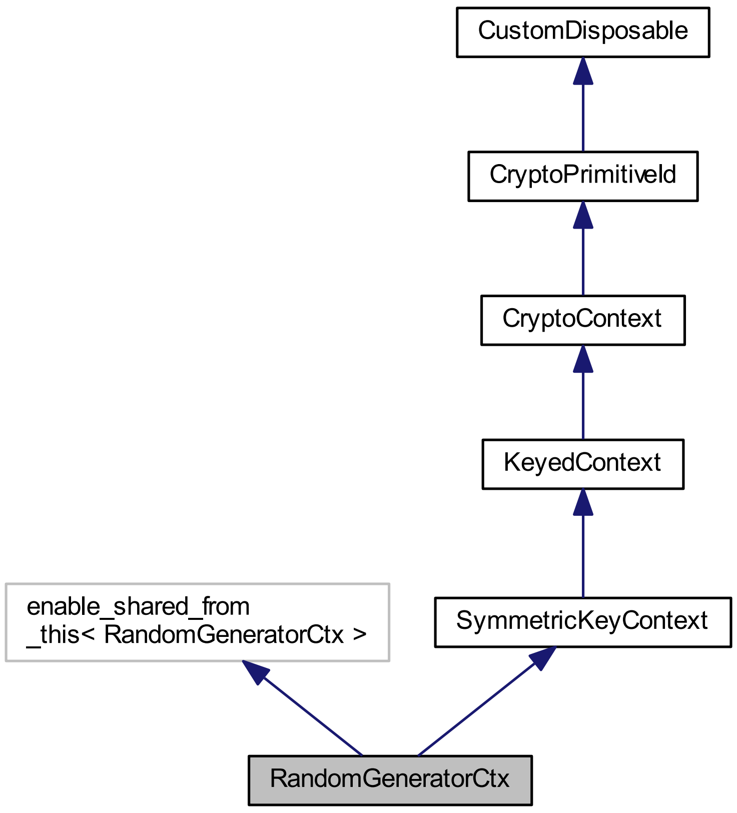
Exceptions

CryptoInvalidArgument	if values of provided key or domain parameters are incompatible with algorithm of this private key context.
CryptoRuntimeError	if provided private key object references to another instance of the domain parameters, i.e. if the `COUID` of provided domain parameters is not equal to the `COUID` referenced from the private key object.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of provided key object.

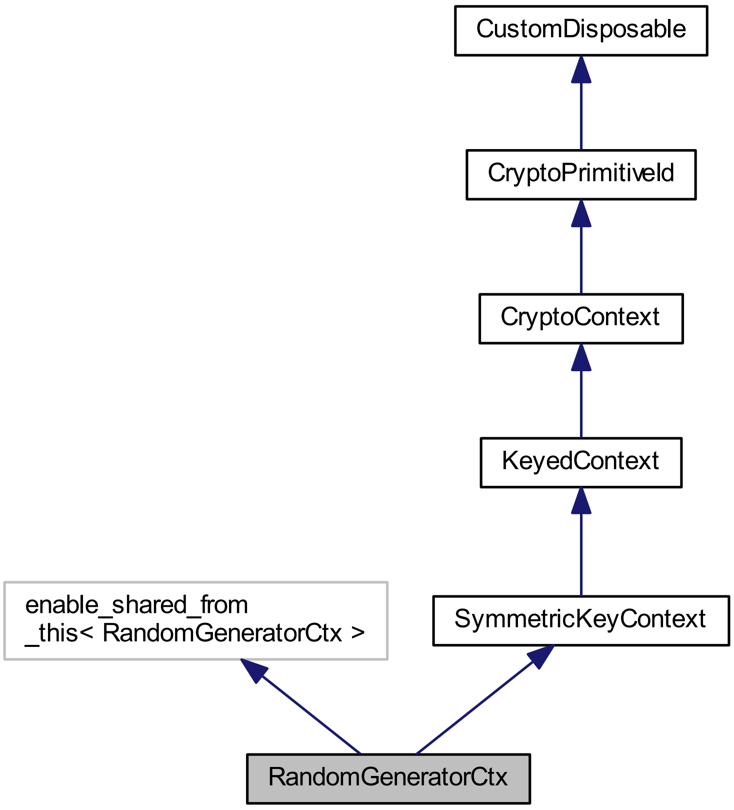
⌋*(RS_CRYPTO_02002, RS_CRYPTO_02003, RS_CRYPTO_02108)*

9.50 RandomGeneratorCtx Interface Reference

Inheritance diagram for RandomGeneratorCtx:



Collaboration diagram for RandomGeneratorCtx:



Public Types

- typedef std::shared_ptr< RandomGeneratorCtx > sptr_t

Public Member Functions

- virtual bool reseed (ReadOnlyMemRegion seed) noexcept(true)=0
- virtual bool reseed (const SecretSeed &seed) noexcept(false)=0
- virtual void generate (WritableMemRegion output) noexcept(true)=0

Additional Inherited Members

9.50.1 Detailed Description

[SWS_CRYPTO_22900] Interface definition: ⌈ Random Number Generator (RNG) Context interface

Note

> Any user of this interface should create shared pointers to it only by calls of the method `shared_from_this()`!

⌋*(RS_CRYPTO_02206)*

9.50.2 Member Typedef Documentation

9.50.2.1 typedef std::shared_ptr<RandomGeneratorCtx> sptr_t

[SWS_CRYPTO_22901] Interface definition: ⌈ Shared smart pointer of the interface.

⌋*(RS_CRYPTO_02311)*

9.50.3 Member Function Documentation

9.50.3.1 virtual bool reseed (ReadOnlyMemRegion *seed*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22911] Interface definition: ⌈ Reseed or update internal state of the RNG by additional entropy.

Parameters

`in`	*seed*	A memory region with the seed value.

Returns

> `true` if the method is supported.

⌋*(RS_CRYPTO_02311)*

9.50.3.2 virtual bool reseed (const SecretSeed & *seed*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22912] Interface definition: ⌈ Reseed or update internal state of the RNG by additional entropy.

Parameters

`in`	*seed*	A memory region with the seed value.

Returns

`true` if the method is supported.

Exceptions

CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `seed` object.

⌋(*RS_CRYPTO_02311*)

9.50.3.3 virtual void generate (WritableMemRegion *output*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_22913] Interface definition: ⌈ Fills whole provided buffer by generated random sequence.

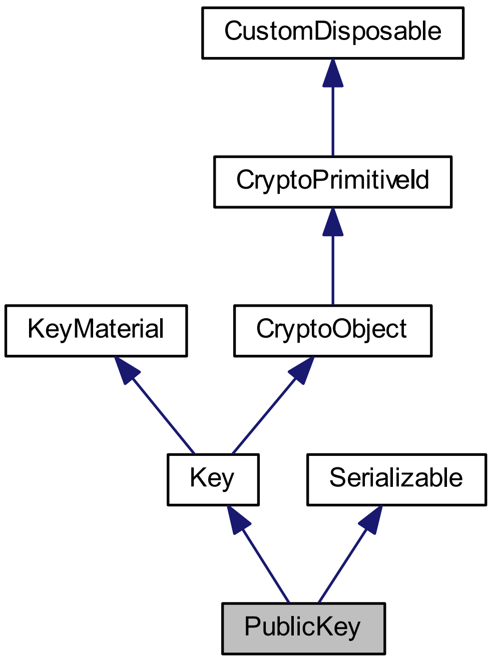
Parameters

`out`	*output*	A target buffer for filling by the generated random sequence.

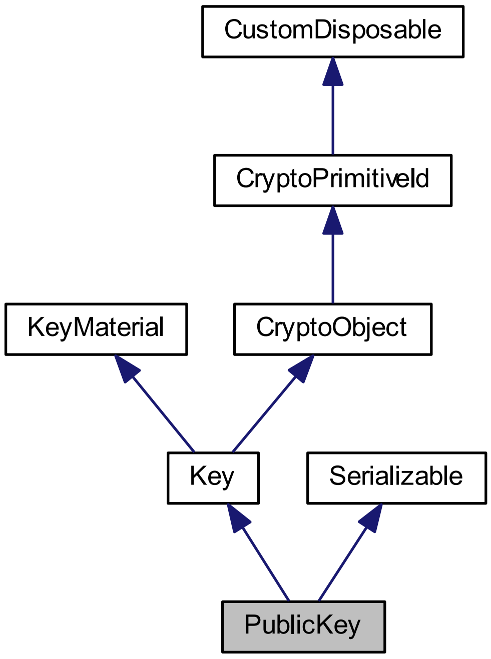
⌋(*RS_CRYPTO_02311*)

9.51 PublicKey Interface Reference

Inheritance diagram for PublicKey:



Collaboration diagram for PublicKey:



Public Types

- typedef std::unique_ptr< const PublicKey, CustomDeleter > uptrc_t

Public Member Functions

- virtual bool checkKey (bool strongCheck=true) const noexcept(true)=0
- virtual std::size_t hashPublicKey (WritableMemRegion hash, HashFunctionCtx &hashFunc) const noexcept(false)=0
- template<typename Alloc >
  void hashPublicKey (std::vector< byte_t, Alloc > &hash, HashFunctionCtx &hashFunc) const noexcept(false)

Additional Inherited Members

9.51.1 Detailed Description

[SWS_CRYPTO_22700] Interface definition: ⌈ General Asymmetric Public Key interface

⌋*(RS_CRYPTO_02403)*

9.51.2 Member Typedef Documentation

9.51.2.1 typedef std::unique_ptr<const PublicKey, CustomDeleter> uptrc_t

[SWS_CRYPTO_22701] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.51.3 Member Function Documentation

9.51.3.1 virtual bool checkKey (bool *strongCheck* = `true`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22711] Interface definition: ⌈ Checks the key for its correctness.

Parameters

`in`	*strongCheck*	Indicates type of required check: strong (if `true`) or fast (if `false`).

Returns

`true` if the key is correct.

⌋*(RS_CRYPTO_02311)*

9.51.3.2 virtual std::size_t hashPublicKey (WritableMemRegion *hash*, HashFunctionCtx & *hashFunc*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22712] Interface definition: ⌈ Calculate hash of the Public Key value.

Parameters

`out`	*hash*	A buffer preallocated for the resulting hash value.
`in`	*hashFunc*	A hash-function instance that should be used the hashing.

Returns

Actual size of the hash value stored to the output buffer.

Note

The original public key value BLOB is available via the Serializable interface.

Exceptions

CryptoInvalidArgument	if size of the hash buffer is not enough for storing of the result.

⌋(*RS_CRYPTO_02311*)

9.51.3.3 void hashPublicKey (std::vector< byte_t, Alloc > & *hash*, HashFunctionCtx & *hashFunc*) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_22713] Interface definition: ⌈ Calculate hash of the Public Key value.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*hash*	Pre-reserved managed container for the resulting hash value.
`in`	*hashFunc*	A hash-function instance that should be used the hashing.

Note

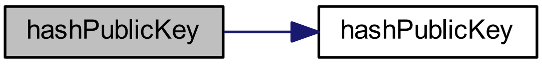
This method sets the size of the output container according to actually saved value!
The original public key value BLOB is available via the Serializable interface.

Exceptions

CryptoInvalidArgument	if capacity of the hash buffer is not enough for storing of the result.

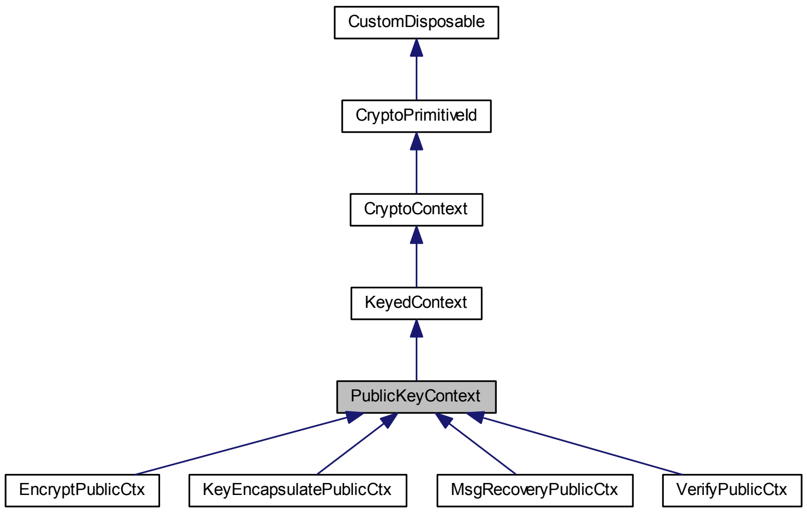
⌋(*RS_CRYPTO_02311*)

Here is the call graph for this function:

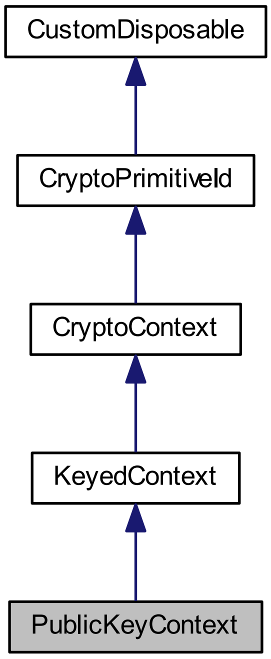


9.52 PublicKeyContext Interface Reference

Inheritance diagram for PublicKeyContext:



Collaboration diagram for PublicKeyContext:



Public Member Functions

- virtual void setKey (const PublicKey &key, DomainParameters::sptrc_t params=nullptr) noexcept(false)=0

Additional Inherited Members

9.52.1 Detailed Description

[SWS_CRYPTO_22800] Interface definition: ⌈ Generalized interface of Public Key algorithm Context

⌋*(RS_CRYPTO_02202)*

9.52.2 Member Function Documentation

9.52.2.1 virtual void setKey (const PublicKey & *key*, DomainParameters::sptrc_t *params* = `nullptr`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_22811] Interface definition: ⌈ Sets (load) a key to the algorithm context.

Parameters

`in`	*key*	A reference to a source key.
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the transformation.

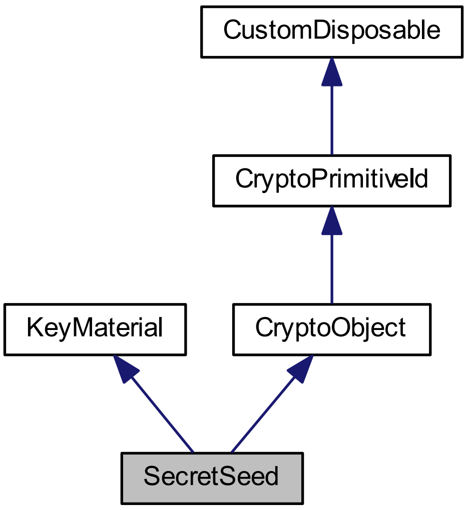
Exceptions

CryptoRuntimeError	if the provided key or domain parameters are incompatible with this public key context.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of provided key object.

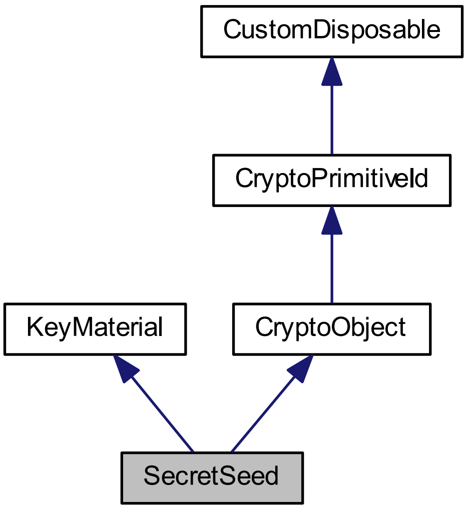
⌋*(RS_CRYPTO_02108)*

9.53 SecretSeed Interface Reference

Inheritance diagram for SecretSeed:



Collaboration diagram for SecretSeed:



Public Types

- typedef std::unique_ptr< const SecretSeed, CustomDeleter > uptrc_t
- using Usage_t = AllowedUsageFlags_t

Additional Inherited Members

9.53.1 Detailed Description

[SWS_CRYPTO_23000] Interface definition: ⌈ Secret Seed object interface

Note

> This object contains a raw bit sequence of specific length (without any filtering of allowed/disallowed values)!
> The secret seed value can be loaded only to a non-key input of a cryptographic transformation context (like IV/salt/nonce)!
> Bit length of the secret seed is specific to concret crypto algorithm and corresponds to maximum of its input/output/salt block-length.

⌋*(RS_CRYPTO_02007)*

9.53.2 Member Typedef Documentation

9.53.2.1 typedef std::unique_ptr<const SecretSeed, CustomDeleter> uptrc_t

[SWS_CRYPTO_23001] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

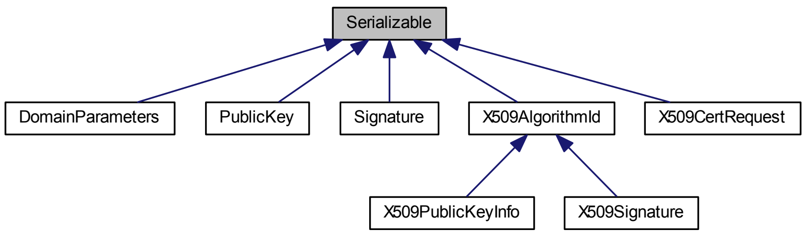
9.53.2.2 using Usage_t = AllowedUsageFlags_t

[SWS_CRYPTO_23002] Interface definition: ⌈ Alias to the container type for bit-flags of allowed usages of the key.

⌋*(RS_CRYPTO_02008)*

9.54 Serializable Interface Reference

Inheritance diagram for Serializable:



Public Types

- typedef std::uint32_t FormatId_t

Public Member Functions

- virtual std::size_t exportPublicly (WritableMemRegion ∗output=nullptr, FormatId_t formatId=FORMAT_DEFAULT) const noexcept(false)=0
- template<typename Alloc >
  void exportPublicly (std::vector< byte_t, Alloc > &output, FormatId_t formatId=FORMAT_DEFAULT) const noexcept(false)

Protected Member Functions

- virtual ∼Serializable ()=default

9.54.1 Detailed Description

[SWS_CRYPTO_23100] Interface definition: ⌈ Serializable object interface

⌋*(RS_CRYPTO_02105)*

9.54.2 Member Typedef Documentation

9.54.2.1 typedef std::uint32_t FormatId_t

[SWS_CRYPTO_23101] Interface definition: ⌈ A container type for bit-flags of allowed usages of the key.

⌋*(RS_CRYPTO_02311)*

9.54.3 Constructor & Destructor Documentation

9.54.3.1 virtual ∼Serializable ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_23110] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.54.4 Member Function Documentation

9.54.4.1 virtual std::size_t exportPublicly (WritableMemRegion ∗ *output* = *`nullptr`*, FormatId_t *formatId* = FORMAT_DEFAULT) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23111] Interface definition: ⌈ Serialize itself publicly.

Parameters

`out`	*output*	Pointer to preallocated output buffer, it can be null if only the required size is interested.
`in`	*formatId*	A Crypto Provider specific identifier of the output format.

Returns

Size required for storing of the output object.

Exceptions

CryptoInvalidArgument	if `(output != nullptr)` and it's capacity is less than required or if the specified format ID is not supported for the object.

⌋*(RS_CRYPTO_02112)*

9.54.4.2 void exportPublicly (std::vector< byte_t, Alloc > & *output*, FormatId_t *formatId* = FORMAT_DEFAULT) const [inline], [noexcept]

[SWS_CRYPTO_23112] Interface definition: ⌈ Serialize itself publicly.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*output*	Pre-reserved managed container for the serialization output.
`in`	*formatId*	A Crypto Provider specific identifier of the output format.

Note

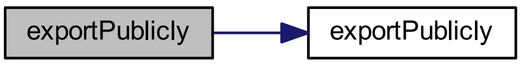
This method sets the size of the output container according to actually saved value!

Exceptions

CryptoInvalidArgument	if capacity of the output buffer is less than required or if the specified format ID is not supported for the object.

⌋*(RS_CRYPTO_02112)*

Here is the call graph for this function:



9.54.5 Member Data Documentation

9.54.5.1 const FormatId_t FORMAT_DEFAULT = 0L [static]

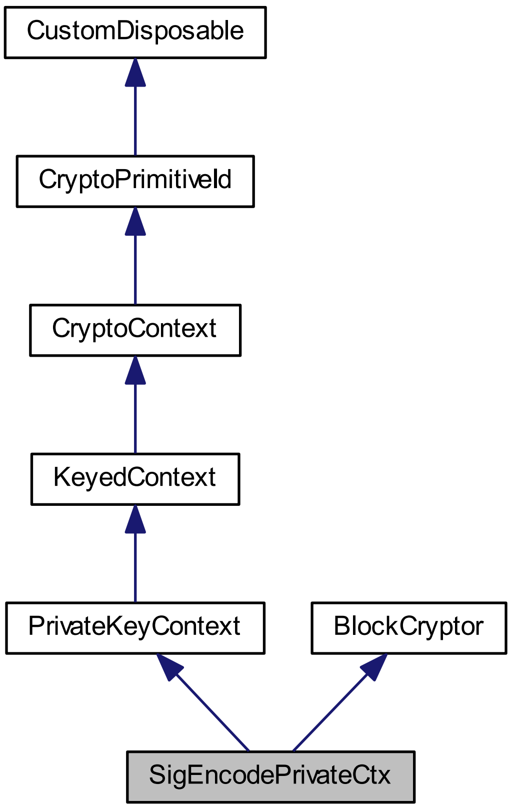
Default format.

9.54.5.2 const FormatId_t FORMAT_RAW_VALUE_ONLY = 1L `[static]`

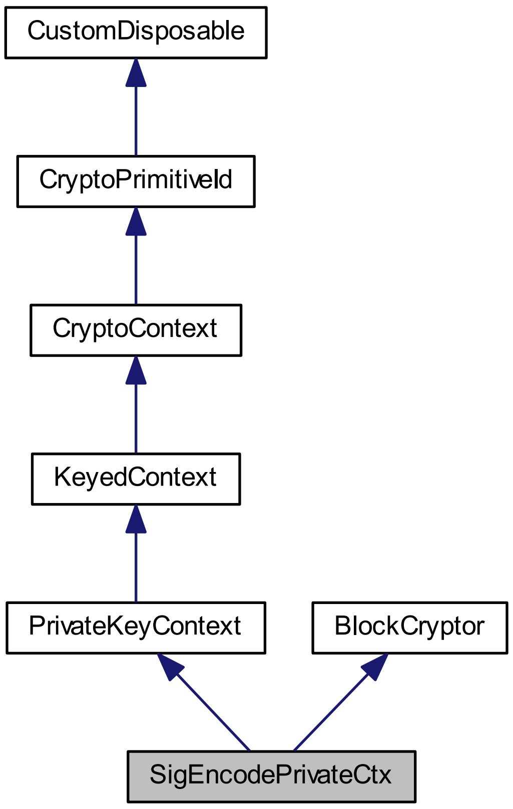
Export only raw value of an object.

9.55 SigEncodePrivateCtx Interface Reference

Inheritance diagram for SigEncodePrivateCtx:



Collaboration diagram for SigEncodePrivateCtx:



Public Types

- typedef std::unique_ptr< SigEncodePrivateCtx, CustomDeleter > uptr_t

Additional Inherited Members

9.55.1 Detailed Description

[SWS_CRYPTO_23200] Interface definition: ⌈ A private key context for asymmetric signature calculation and short message encoding (RSA-like).

Note

> Restricted groups of trusted subscribers can use this primitive for simultaneous provisioning of confidentiality, authenticity and non-repudiation of short messages, if the public key is generated appropriately and kept in secret.

⌋*(RS_CRYPTO_02202, RS_CRYPTO_02204)*

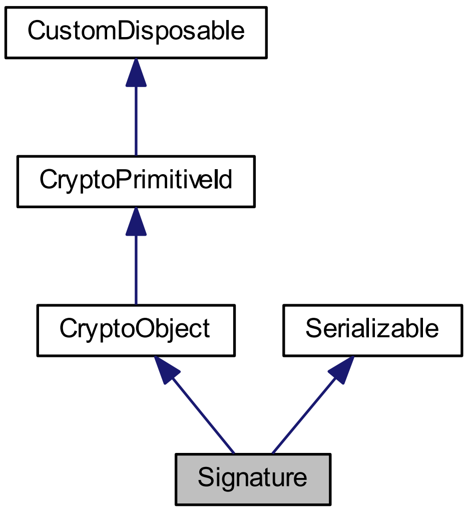
9.55.2 Member Typedef Documentation

9.55.2.1 typedef std::unique_ptr<SigEncodePrivateCtx, CustomDeleter> uptr_t

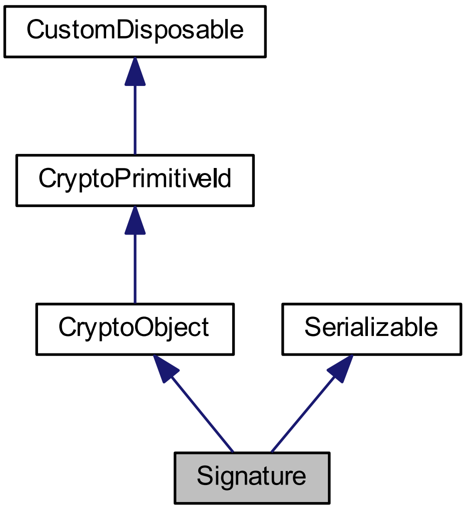
[SWS_CRYPTO_23201] Interface definition: ⌈ Unique smart pointer of the interface.
⌋*(RS_CRYPTO_02404)*

9.56 Signature Interface Reference

Inheritance diagram for Signature:



Collaboration diagram for Signature:



Public Types

- typedef std::unique_ptr< const Signature, CustomDeleter > uptrc_t

Public Member Functions

- virtual CryptoPrimitiveId::AlgId_t getHashAlgId () const noexcept(true)=0

Protected Member Functions

- virtual ~Signature ()=default

Additional Inherited Members

9.56.1 Detailed Description

[SWS_CRYPTO_23300] Interface definition: ⌈ Signature container interface

Note

> This interface is applicable for keeping the Digital Signature, Hash Digest, (Hash-based) Message Authentication Code (MAC/HMAC).
> In case of a keyed signature (Digital Signature or MAC/HMAC) a `COUID` of the signature verification key can be obtained by a call of `CryptoObject::hasDependence()`!

⌋(*RS_CRYPTO_02203*, *RS_CRYPTO_02204*, *RS_CRYPTO_02205*)

9.56.2 Member Typedef Documentation

9.56.2.1 typedef std::unique_ptr<const Signature, CustomDeleter> uptrc_t

[SWS_CRYPTO_23301] Interface definition: ⌈ Unique smart pointer of the interface.

⌋(*RS_CRYPTO_02404*)

9.56.3 Constructor & Destructor Documentation

9.56.3.1 virtual ∼Signature ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_23310] Interface definition: ⌈ Destructor.

⌋(*RS_CRYPTO_02311*)

9.56.4 Member Function Documentation

9.56.4.1 virtual CryptoPrimitiveId::AlgId_t getHashAlgId ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23311] Interface definition: ⌈ Returns an ID of hash algorithm used for this signature object production.

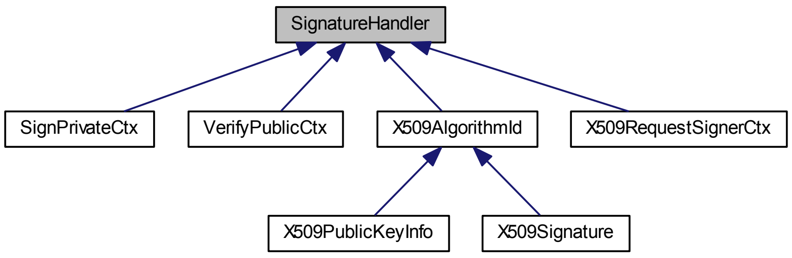
Returns

ID of used hash algorithm only (without signature algorithm specification).

⌋(*RS_CRYPTO_02311*)

9.57 SignatureHandler Interface Reference

Inheritance diagram for SignatureHandler:



Public Member Functions

- virtual std::size_t requiredHashSize () const noexcept(true)=0
- virtual CryptoPrimitiveId::AlgId_t requiredHashAlgId () const noexcept(true)=0
- virtual std::size_t signatureSize () const noexcept(true)=0

Protected Member Functions

- virtual ∼SignatureHandler ()=default

9.57.1 Detailed Description

[SWS_CRYPTO_23400] Interface definition: ⌈ A basic interface for both types of the signature handlers: signer and verifier.

⌋(*RS_CRYPTO_02204*)

9.57.2 Constructor & Destructor Documentation

9.57.2.1 virtual ~SignatureHandler ( ) `[protected]`, `[virtual]`, `[default]`

[SWS_CRYPTO_23410] Interface definition: ⌈ Destructor.

⌋*(RS_CRYPTO_02311)*

9.57.3 Member Function Documentation

9.57.3.1 virtual std::size_t requiredHashSize ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23411] Interface definition: ⌈ Returns a hash size required by current signature algorithm.

Returns

Required hash size in bytes.

⌋*(RS_CRYPTO_02309)*

9.57.3.2 virtual CryptoPrimitiveId::AlgId_t requiredHashAlgId ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23412] Interface definition: ⌈ Returns an ID of hash algorithm required by current signature algorithm.

Returns

Required hash algorithm ID or `ALGID_ANY` if the signature algorithm specification does not include a concrete hash function.

⌋*(RS_CRYPTO_02309)*

9.57.3.3 virtual std::size_t signatureSize ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23413] Interface definition: ⌈ Returns size of the signature value produced and required by the current algorithm.

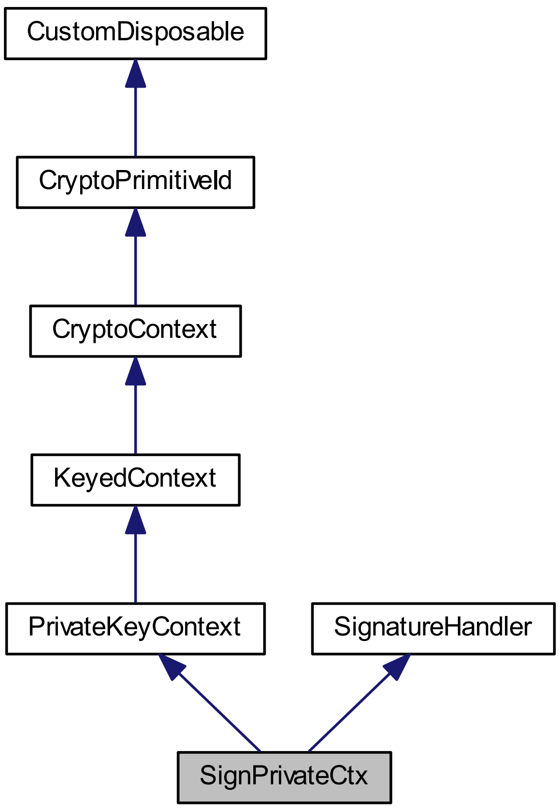
Returns

Size of the signature value in bytes.

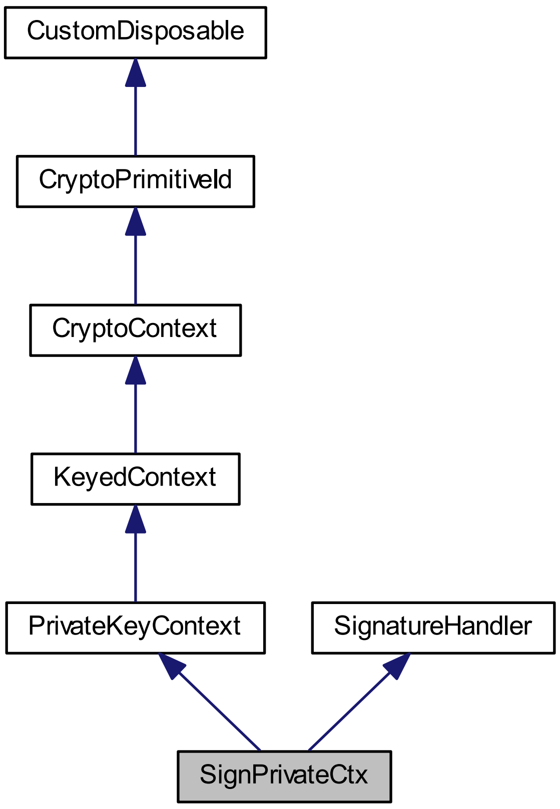
⌋ *(RS_CRYPTO_02309)*

9.58 SignPrivateCtx Interface Reference

Inheritance diagram for SignPrivateCtx:



Collaboration diagram for SignPrivateCtx:



Public Types

- typedef std::unique_ptr< SignPrivateCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual Signature::uptrc_t sign (const HashFunctionCtx &hash, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual std::size_t sign (WritableMemRegion signature, ReadOnlyMemRegion hash) const noexcept(false)=0
- template< typename Alloc >
  void sign (std::vector< byte_t, Alloc > &signature, ReadOnlyMemRegion hash) const noexcept(false)

Additional Inherited Members

9.58.1 Detailed Description

[SWS_CRYPTO_23500] Interface definition: ⌈ Signature Private key Context interface

⌋ *(RS_CRYPTO_02204)*

9.58.2 Member Typedef Documentation

9.58.2.1 typedef std::unique_ptr<SignPrivateCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_23501] Interface definition: ⌈ Unique smart pointer of the interface.

⌋ *(RS_CRYPTO_02404)*

9.58.3 Member Function Documentation

9.58.3.1 virtual Signature::uptrc_t sign (const HashFunctionCtx & *hash*, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23511] Interface definition: ⌈ Sign a provided digest value stored in the hash-function context.

Parameters

`in`	*hash*	A finalized hash-function context that contains a digest value ready for sign.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to serialized signature.

Note

This method must put the hash-function algorithm ID and a COUID of the used key-pair to the resulting signature object!

Exceptions

CryptoInvalidArgument	if hash-function algorithm does not comply with the signature algorithm specification of the context.
CryptoLogicError	if the method `hash.finish()` was not called before the call of this method or this context was not initialized by a key value.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02404*)

9.58.3.2 virtual std::size_t sign (WritableMemRegion *signature*, ReadOnlyMemRegion *hash*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23512] Interface definition: ⌈ Sign a directly provided hash value.

Parameters

`out`	*signature*	A buffer for resulting signature.
`in`	*hash*	A hash value that should be signed.

Returns

Actual size of the signature value stored to the output buffer.

Exceptions

CryptoInvalidArgument	if sizes of the input hash or resulting buffer are incorrect.
CryptoLogicError	if the context was not initialized by a key value.

⌋(*RS_CRYPTO_02404*)

9.58.3.3 void sign (std::vector< byte_t, Alloc > & *signature*, ReadOnlyMemRegion *hash*) const `[inline]`, `[noexcept]`

[SWS_CRYPTO_23513] Interface definition: ⌈ Sign a directly provided hash value.

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*signature*	Pre-reserved managed container for resulting signature.
`in`	*hash*	A hash value that should be signed.

Note

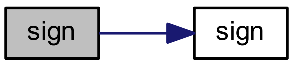
This method sets the size of the output container according to actually saved value!

Exceptions

CryptoInvalidArgument	if size of the input hash value is incorrect or capacity of the output container is not enough.
CryptoLogicError	if the context was not initialized by a key value.

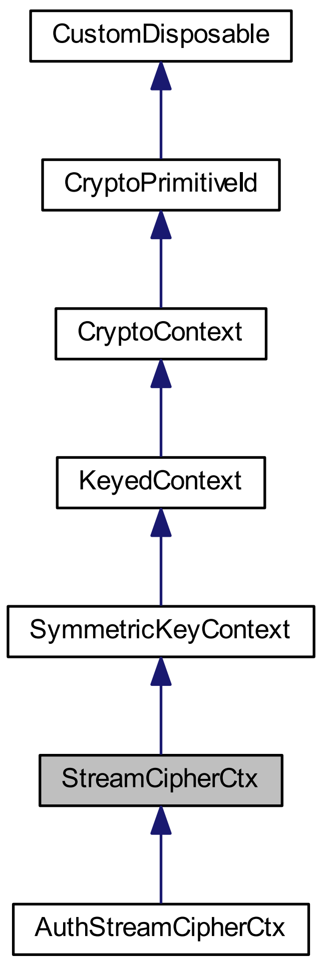
⌋(*RS_CRYPTO_02404*)

Here is the call graph for this function:

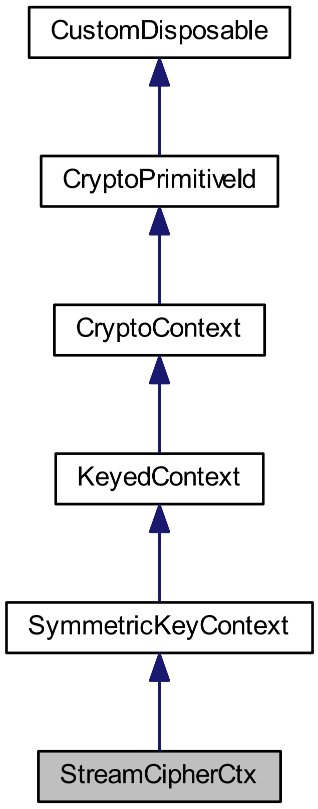


9.59 StreamCipherCtx Interface Reference

Inheritance diagram for StreamCipherCtx:



Collaboration diagram for StreamCipherCtx:



Public Types

- typedef std::unique_ptr< StreamCipherCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t getIVSize () const noexcept(true)=0
- virtual std::size_t getBlockSize () const noexcept(true)=0
- virtual bool isBytewiseMode () const noexcept(true)=0
- virtual bool isSeekableMode () const noexcept(true)=0
- virtual bool isValidIVSize (std::size_t ivSize) const noexcept(true)=0
- virtual void start (ReadOnlyMemRegion iv) noexcept(false)=0
- virtual void start (const SecretSeed &iv) noexcept(false)=0

- virtual void seek (std::int64_t offset, bool fromBegin=true) noexcept(false)=0
- virtual void processBlocks (WritableMemRegion out, ReadOnlyMemRegion in) noexcept(false)=0
- virtual void processBlocks (ReadWriteMemRegion inOut) noexcept(false)=0
- virtual std::size_t processBytes (WritableMemRegion out, ReadOnlyMemRegion in) noexcept(false)=0
- template<typename Alloc >
  void processBytes (std::vector< byte_t, Alloc > &out, ReadOnlyMemRegion in) noexcept(false)
- virtual std::size_t finishBytes (WritableMemRegion out, ReadOnlyMemRegion in) noexcept(false)=0
- template<typename Alloc >
  void finishBytes (std::vector< byte_t, Alloc > &out, ReadOnlyMemRegion in) noexcept(false)

Additional Inherited Members

9.59.1 Detailed Description

[SWS_CRYPTO_23600] Interface definition: ⌈ Generalized Stream Cipher Context interface (cover all modes of operation)

⌋*(RS_CRYPTO_02201)*

9.59.2 Member Typedef Documentation

9.59.2.1 typedef std::unique_ptr<StreamCipherCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_23601] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.59.3 Member Function Documentation

9.59.3.1 virtual std::size_t getIVSize ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23611] Interface definition: ⌈ Returns default expected size of Initialization Vector (IV).

Returns

Default expected size of IV in bytes.

⌋(*RS_CRYPTO_02309*)

9.59.3.2 virtual std::size_t getBlockSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23612] Interface definition: ⌈ Returns block size of base algorithm.

Returns

Size of block in bytes.

⌋(*RS_CRYPTO_02309*)

9.59.3.3 virtual bool isBytewiseMode ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23613] Interface definition: ⌈ Checks the operation mode for the bytewise property.

Returns

`true` if the mode can process a message the byte by byte (without padding) or `false` if only the block by block (with padding).

⌋(*RS_CRYPTO_02309*)

9.59.3.4 virtual bool isSeekableMode ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23614] Interface definition: ⌈ Checks if the seek operation is supported in the current mode.

Returns

`true` the seek operation is supported in the current mode or `false` otherwise.

⌋(*RS_CRYPTO_02309*)

9.59.3.5 virtual bool isValidIVSize (std::size_t *ivSize*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23615] Interface definition: ⌈ Verifies validity of specific IV length.

Parameters

`in`	*ivSize*	Length of the IV in bytes.

Returns

true if provided IV length is supported by the algorithm.

⌋(*RS_CRYPTO_02309*)

9.59.3.6 virtual void start (ReadOnlyMemRegion *iv*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_23616] Interface definition: ⌈ Initialize stream cipher for a new encryption/decryption session.

Parameters

`in`	*iv*	A buffer containing the Initialization Vector.

Exceptions

CryptoLogicError	if the context was not initialized by a key value.
CryptoInvalidArgument	if the size of provided Initialization Vector is not enough for initialization.

⌋(*RS_CRYPTO_02302*)

9.59.3.7 virtual void start (const SecretSeed & *iv*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_23617] Interface definition: ⌈ Initialize stream cipher for a new encryption/decryption session.

Parameters

`in`	*iv*	A buffer containing the Initialization Vector.

Exceptions

CryptoLogicError	if the context was not initialized by a key value.

CryptoInvalidArgument	if the size of provided Initialization Vector is not enough for initialization.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of the provided `iv` object.

⌋(*RS_CRYPTO_02302*)

9.59.3.8 virtual void seek (std::int64_t *offset*, bool *fromBegin* = `true`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23618] Interface definition: ⌈ Sets the position of the next byte within the stream of the encryption/decryption gamma.

Parameters

`in`	*offset*	Offset value in bytes, relative to begin or current position in the gamma stream.
`in`	*fromBegin*	Define a starting point for positioning within the stream: from begin (if `true`) or from current position (if `false`).

Returns

`true` if the seek operation is supported by the current mode and value of the offset is correct (e.g. CTR or GCM modes).

Exceptions

CryptoLogicError	if the seek operation is not supported by the current mode or data processing was not started by the `start()` method call.
CryptoInvalidArgument	if value of the offset is incorrect (i.e. points before begin, it is an optional error condition).

⌋(*RS_CRYPTO_02304*)

9.59.3.9 virtual void processBlocks (WritableMemRegion *out*, ReadOnlyMemRegion *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23619] Interface definition: ⌈ Processes initial parts of message aligned to the block-size boundary.

Parameters

`out`	*out*	An output data buffer.
`in`	*in*	An input data buffer.

Note

It is a copy-optimized method that doesn't use the internal cache buffer! It can be used only before processing of any non-aligned to the block-size boundary data.

Exceptions

CryptoInvalidArgument	if sizes of input and output buffers are not equal or not divisible on the block size (see `getBlockSize()`).
CryptoLogicError	if this method called after processign of non-aligned data (to the block-size boundary) or data processing was not started by the `start()` method call.

⌋(*RS_CRYPTO_02302*)

9.59.3.10 virtual void processBlocks (ReadWriteMemRegion *inOut*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23620] Interface definition: ⌈ Processes initial parts of message aligned to the block-size boundary.

Parameters

`in,out`	*inOut*	An input and output data buffer, i.e. whole buffer will be updated.

Note

The "inOut" buffer MUST have a size divisible by the block size (see `getBlockSize()`).
It is a copy-optimized method that doesn't use internal cache buffer! It can be used up to first non-block aligned data processing.

Exceptions

CryptoInvalidArgument	if size of the inOut buffer is not divisible on the block size (see `getBlockSize()`).
CryptoLogicError	if this method called after processign of non-aligned data (to the block-size boundary) or data processing was not started by the `start()` method call.

⌋(*RS_CRYPTO_02302*)

9.59.3.11 virtual std::size_t processBytes (WritableMemRegion *out*, ReadOnlyMemRegion *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23621] Interface definition: ⌈ Processes a non-final part of message (that is not aligned to the block-size boundary).

Parameters

`out`	*out*	An output data buffer.
`in`	*in*	An input data buffer.

Returns

Actual size of output data (i.e. a number of first bytes updated in the output buffer).

Note

If `(isBytewiseMode() == false)` then it **must** be: `out.size() >= (((in.size() + getBlockSize() - 1) / getBlockSize()) * getBlockSize())`
If `(isBytewiseMode() == true)` then it **must** be: `out.size() >= in.size()`
This method is "copy inefficient", therefore it should be used only by a calling application that cannot control the chunking of the original message!

Exceptions

CryptoInvalidArgument	if the output buffer has incorrect size.
CryptoLogicError	if data processing was not started by the `start()` method call.

⌋(*RS_CRYPTO_02302*)

9.59.3.12 void processBytes (std::vector< byte_t, Alloc > & *out,* ReadOnlyMemRegion *in*) `[inline]`, `[noexcept]`

[SWS_CRYPTO_23622] Interface definition: ⌈ Processes a non-final part of message (that is not aligned to the block-size boundary).

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*out*	A managed container for output data.
`in`	*in*	An input data buffer.

Note

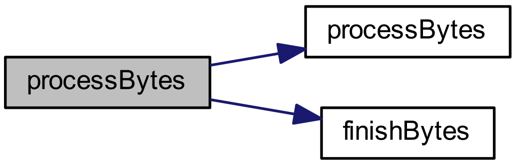
This method sets the size of the output container according to actually saved value.
If `(isBytewiseMode() == false)` then it **must** be: `out.capacity() >= (((in.size() + getBlockSize() - 1) / getBlockSize()) * getBlockSize())`
If `(isBytewiseMode() == true)` then it **must** be: `out.capacity() >= in.size()`
This method is "copy inefficient", therefore it should be used only by a calling application that cannot control the chunking of the original message!

Exceptions

CryptoInvalidArgument	if capacity of the output container is not enough.
CryptoLogicError	if data processing was not started by the `start()` method call.

⌋*(RS_CRYPTO_02302)*

Here is the call graph for this function:



9.59.3.13 virtual std::size_t finishBytes (WritableMemRegion *out*, ReadOnlyMemRegion *in*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_23623] Interface definition: ⌈ Processes a final part of message (that is not aligned to the block-size boundary).

Parameters

`out`	*out*	An output data buffer.
`in`	*in*	An input data buffer.

Returns

Actual size of output data (i.e. a number of first bytes that were updated in the output buffer).

Note

If `(isBytewiseMode() == false)` then it **must** be: `out.size() >= (((in.size() + getBlockSize() * (isDirectTransform() ? 2 : 1) - 1) / getBlockSize()) * getBlockSize())`
If `(isBytewiseMode() == true)` then it **must** be: `out.size() >= in.size()`

Exceptions

CryptoInvalidArgument	if the output buffer has incorrect size.
CryptoLogicError	if data processing was not started by the `start()` method call.

⌋*(RS_CRYPTO_02302)*

9.59.3.14 void finishBytes (std::vector< byte_t, Alloc > & *out*, ReadOnlyMemRegion *in*) `[inline]`, `[noexcept]`

[SWS_CRYPTO_23624] Interface definition: ⌈ Processes a final part of message (that is not aligned to the block-size boundary).

Template Parameters

Alloc	Custom allocator type of the output container.

Parameters

`out`	*out*	A managed container for output data.
`in`	*in*	An input data buffer.

Note

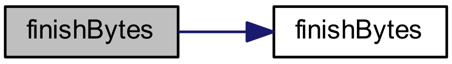
This method sets the size of the output container according to actually saved value.
If `(isBytewiseMode() == false)` then it **must** be: `out.capacity() >= (((in.size() + getBlockSize() * (isDirectTransform() ? 2 : 1) - 1) / getBlockSize()) * getBlockSize())`
If `(isBytewiseMode() == true)` then it **must** be: `out.capacity() >= in.size()`

Exceptions

CryptoInvalidArgument	if capacity of the output container is not enough.
CryptoLogicError	if data processing was not started by the `start()` method call.

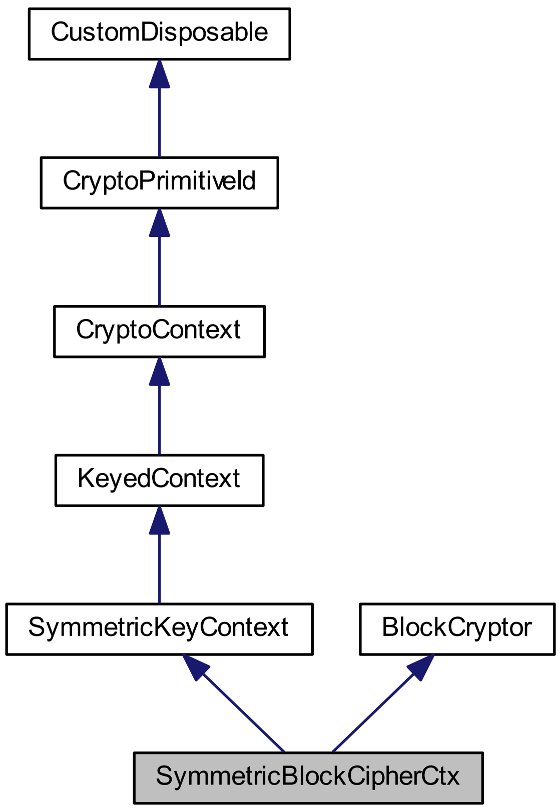
⌋(*RS_CRYPTO_02302*)

Here is the call graph for this function:

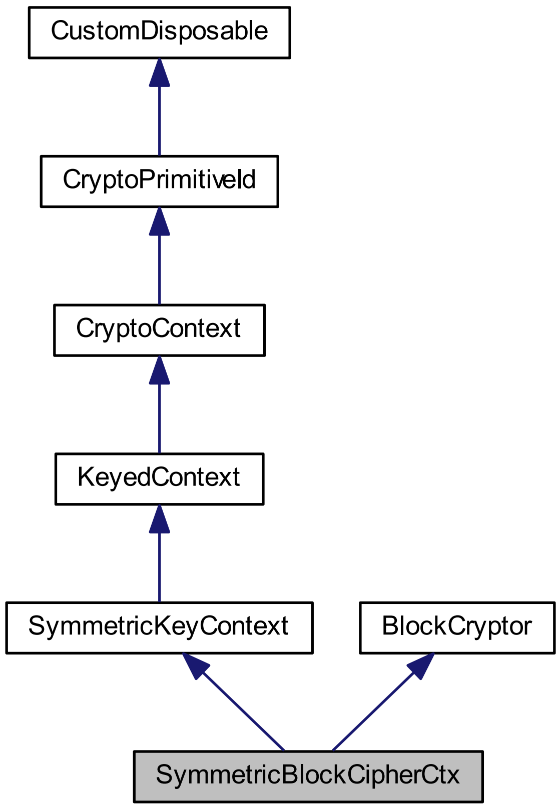


9.60 SymmetricBlockCipherCtx Interface Reference

Inheritance diagram for SymmetricBlockCipherCtx:



Collaboration diagram for SymmetricBlockCipherCtx:



Public Types

- typedef std::unique_ptr< SymmetricBlockCipherCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t getBlockSize () const noexcept(true)=0
- virtual void processBlocks (WritableMemRegion out, ReadOnlyMemRegion in) const noexcept(false)=0

Additional Inherited Members

9.60.1 Detailed Description

[SWS_CRYPTO_23700] Interface definition: ⌈ Interface of a Symmetric Block Cipher Context with padding

⌋*(RS_CRYPTO_02201)*

9.60.2 Member Typedef Documentation

9.60.2.1 typedef std::unique_ptr<SymmetricBlockCipherCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_23701] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.60.3 Member Function Documentation

9.60.3.1 virtual std::size_t getBlockSize ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23711] Interface definition: ⌈ Returns a fixed size of the input/output block of data.

Returns

Size of the data block in bytes.

Note

```
getBlockSize() == BlockCryptor::maxInputSize(true) ==
BlockCryptor::maxOutputSize(true)
```

⌋*(RS_CRYPTO_02302, RS_CRYPTO_02309)*

9.60.3.2 virtual void processBlocks (WritableMemRegion *out*, ReadOnlyMemRegion *in*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23712] Interface definition: ⌈ Processes provided blocks without padding.

Parameters

out	*out*	An output data buffer.
in	*in*	An input data buffer.

Note

The in and out buffers **must** have equal size and this size MUST be divisible by the block size (see getBlockSize()).

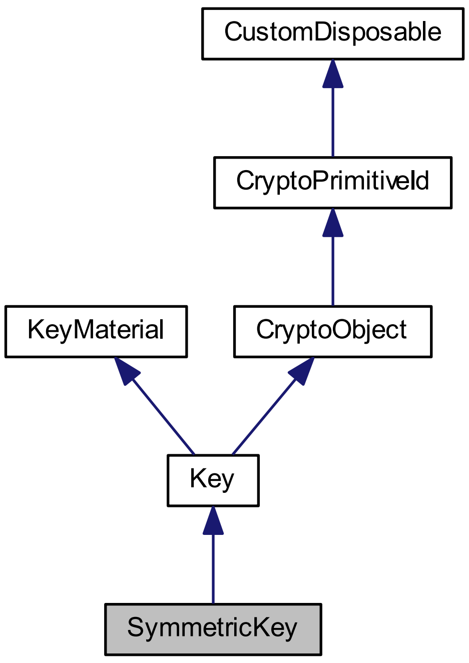
Exceptions

CryptoLogicError	if the context was not initialized by a key value.
CryptoInvalidArgument	if input or output buffers have incorrect sizes.

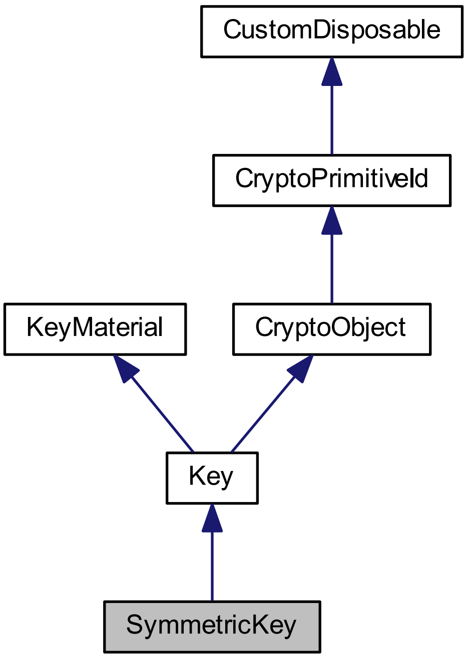
⌋*(RS_CRYPTO_02302)*

9.61 SymmetricKey Interface Reference

Inheritance diagram for SymmetricKey:



Collaboration diagram for SymmetricKey:



Public Types

- typedef std::unique_ptr< const SymmetricKey, CustomDeleter > uptrc_t

Additional Inherited Members

9.61.1 Detailed Description

[SWS_CRYPTO_23800] Interface definition: ⌈ Symmetric Key interface
⌋(*RS_CRYPTO_02001*, *RS_CRYPTO_02403*)

9.61.2 Member Typedef Documentation

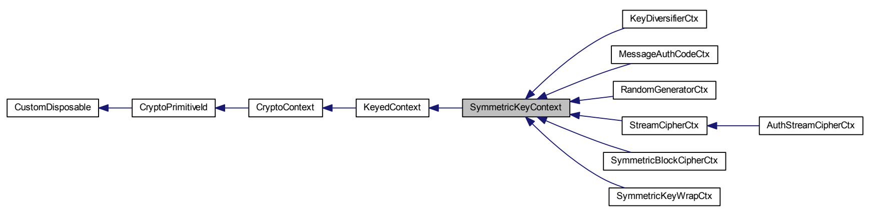
9.61.2.1 typedef std::unique_ptr<const SymmetricKey, CustomDeleter> uptrc_t

[SWS_CRYPTO_23801] Interface definition: ⌈ Unique smart pointer of the interface.

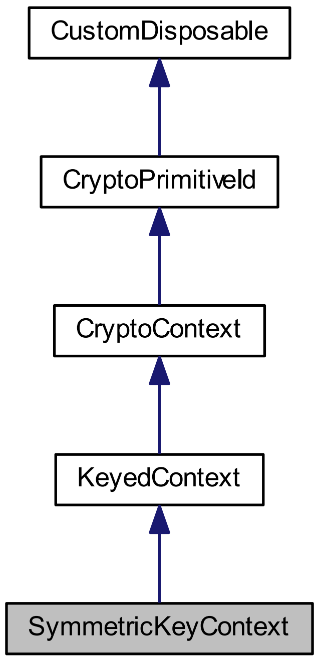
⌋(*RS_CRYPTO_02404*)

9.62 SymmetricKeyContext Interface Reference

Inheritance diagram for SymmetricKeyContext:



Collaboration diagram for SymmetricKeyContext:



Public Member Functions

- virtual void setKey (const SymmetricKey &key, bool directTransform=true) noexcept(false)=0

- virtual bool isDirectTransform () const noexcept(true)=0

Additional Inherited Members

9.62.1 Detailed Description

[SWS_CRYPTO_23900] Interface definition: ⌈ Generalized interface of Symmetric Key algorithm Context

⌋ *(RS_CRYPTO_02201)*

9.62.2 Member Function Documentation

9.62.2.1 virtual void setKey (const SymmetricKey & *key*, bool *directTransform* = *true*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_23911] Interface definition: ⌈ Sets (deploy) a key to the symmetric algorithm context.

Parameters

`in`	*key*	A reference to a source key.
`in`	*directTransform*	Direction indicator: deploy the key for direct transformation (if `true`) or for reverse one (if `false`).

Exceptions

CryptoRuntimeError	if the crypto primitive or usage restictions of the provided key is incompatible with this context or transformation direction.
CryptoUsageViolation	if this transformation type is prohibited by the "allowed usage" restrictions of provided key object.

⌋ *(RS_CRYPTO_02001, RS_CRYPTO_02003)*

9.62.2.2 virtual bool isDirectTransform ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_23912] Interface definition: ⌈ Gets configured "direction" of the transformation: direct (e.g. encryption) or reverse (e.g. decryption).

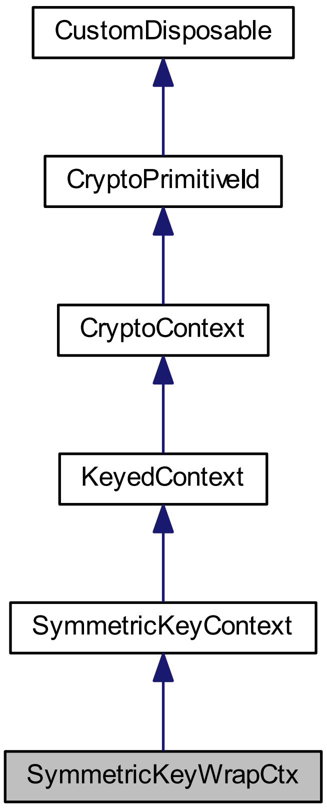
Returns

`true` if the context configured for a direct transformation and `false` if for a reverse one.

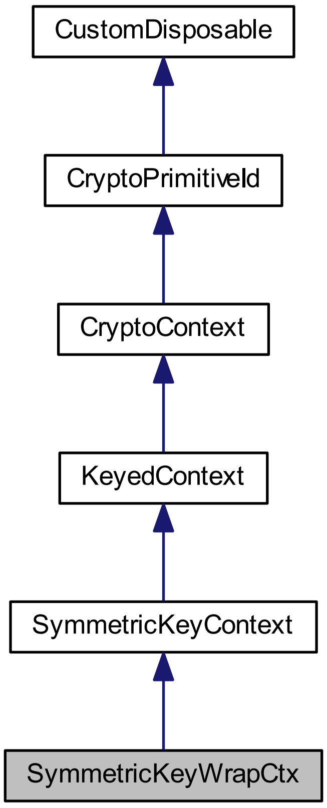
⌋(*RS_CRYPTO_02309*)

9.63 SymmetricKeyWrapCtx Interface Reference

Inheritance diagram for SymmetricKeyWrapCtx:



Collaboration diagram for SymmetricKeyWrapCtx:



Public Types

- typedef std::unique_ptr< SymmetricKeyWrapCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual std::size_t targetKeyGranularity () const noexcept(true)=0
- virtual std::size_t maxTargetKeyLength () const noexcept(true)=0
- virtual std::size_t calculateWrappedKeySize (std::size_t keyLength) const noexcept(true)=0
- virtual void wrapKeyMaterial (WritableMemRegion wrapped, const KeyMaterial &key) const noexcept(false)=0

- virtual SecretSeed::uptrc_t unwrapSeed (ReadOnlyMemRegion wrappedSeed, AlgId_t targetAlgId, SecretSeed::Usage_t allowedUsage, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual Key::uptrc_t unwrapKey (ReadOnlyMemRegion wrappedKey, AlgId_t algId, Key::Usage_t allowedUsage, DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.63.1 Detailed Description

[SWS_CRYPTO_24000] Interface definition: ⌈ Interface of a symmetric key wrap algorithm (for AES it should be compatible with RFC3394 or RFC5649).

Note

> Key Wrapping of a whole key object (including associated meta-data) can be done by `exportXxx()` / `importXxx()` methods, but without compliance to RFC3394 or RFC5649.

⌋*(RS_CRYPTO_02104, RS_CRYPTO_02208, RS_CRYPTO_02404)*

9.63.2 Member Typedef Documentation

9.63.2.1 typedef std::unique_ptr<SymmetricKeyWrapCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_24001] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.63.3 Member Function Documentation

9.63.3.1 virtual std::size_t targetKeyGranularity ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24011] Interface definition: ⌈ Returns expected granularity of the target key (block size).

Returns

> Size of the block in bytes.

Note

> If the class implements RFC3394 (KW without padding) then this method should return 8 (i.e. 8 octets = 64 bits).
> If the class implements RFC5649 (KW with padding) then this method should return 1 (i.e. 1 octet = 8 bits).

⌋(*RS_CRYPTO_02311*)

9.63.3.2 virtual std::size_t maxTargetKeyLength ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24012] Interface definition: ⌈ Returns maximum length of the target key supported by the implementation.

Returns

> Maximum length of the target key in bits.

Note

> This method can be useful for some implementations different from RFC3394 / RFC5649.

⌋(*RS_CRYPTO_02311*)

9.63.3.3 virtual std::size_t calculateWrappedKeySize (std::size_t *keyLength*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24013] Interface definition: ⌈ Calculates size of the wrapped key in bytes from original key length in bits.

Parameters

`in`	*keyLength*	Original key length in bits.

Returns

> Size of the wrapped key in bytes.

Note

This method can be useful for some implementations different from RFC3394 / RFC5649.

⌋(*RS_CRYPTO_02311*)

9.63.3.4 virtual void wrapKeyMaterial (WritableMemRegion *wrapped,* const KeyMaterial & *key*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24014] Interface definition: ⌈ Executes the Key Wrap operation for the provided key material.

Parameters

`out`	*wrapped*	Output buffer for the wrapped key.
`in`	*key*	A key that should be wrapped.

Returns

A smart unique pointer to Secure BLOB that contains wrapped key.

Note

This method should be compliant to RFC3394 or RFC5649 if implementation is based on the AES block cipher and applied to an AES key.
Method `calculateWrappedKeySize()` can be used for size calculation of the required output buffer.

Exceptions

CryptoInvalidArgument	if the size of wrapped buffer is not enough for storing the result or key has unsupported length.
CryptoLogicError	if the context was not initialized by a key value.

⌋(*RS_CRYPTO_02311*)

9.63.3.5 virtual SecretSeed::uptrc_t unwrapSeed (ReadOnlyMemRegion *wrappedSeed,* AlgId_t *targetAlgId,* SecretSeed::Usage_t *allowedUsage,* ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24015] Interface definition: ⌈ Executes the Key Unwrap operation for provided BLOB and produce `SecretSeed` object.

Parameters

`in`	*wrappedSeed*	A memory region that contains wrapped seed.
`in`	*targetAlgId*	Target symmetric algorithm identifier (also defines a target seed-length).
`in`	*allowedUsage*	Allowed usage scope of the target seed.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A smart unique pointer to `SecretSeed` object, which keeps unwrapped key material.

Note

This method should be compliant to RFC3394 or RFC5649 if implementation is based on the AES block cipher and applied to an AES key material.
The created `SecretSeed` object has following attributes: session and non-exportable (because it was imported without meta-information).

Exceptions

CryptoInvalidArgument	if the size of provided wrapped key is unsupported.
CryptoLogicError	if the context was not initialized by a key value.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋*(RS_CRYPTO_02007, RS_CRYPTO_02404)*

9.63.3.6 virtual Key::uptrc_t unwrapKey (ReadOnlyMemRegion *wrappedKey*, AlgId_t *algId*, Key::Usage_t *allowedUsage*, DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24016] Interface definition: ⌈ Executes the Key Unwrap operation for provided BLOB and produce `Key` object.

Parameters

`in`	*wrappedKey*	A memory region that contains wrapped key.
`in`	*algId*	Identifier of target symmetric crypto algorithm.
`in`	*allowedUsage*	Flags that define a list of allowed transformations' types in which the target key can be used (see constants in scope of Key).
`in`	*params*	An optional pointer to Domain Parameters required for full specification of the symmetric transformation (e.g. variable S-boxes of GOST28147-89).
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

A smart unique pointer to `Key` object, which keeps unwrapped key material.

Note

This method should be compliant to RFC3394 or RFC5649 if implementation is based on the AES block cipher and applied to an AES key.
The created `Key` object has following attributes: session and non-exportable (because it was imported without meta-information)!
If `(params != nullptr)` then the domain parameters object must be in the completed state (see DomainParameters)!
If `(params != nullptr)` then at least the parameters' COUID must be saved to the dependency field of the produced key object.

Exceptions

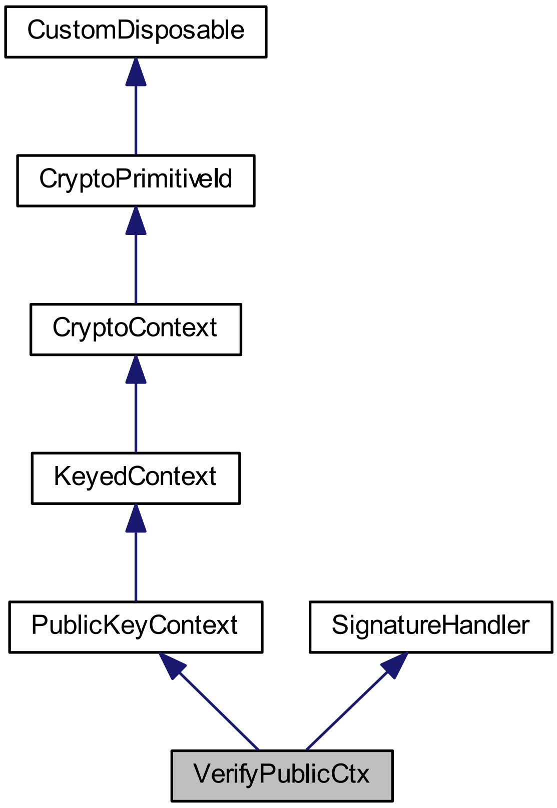
CryptoInvalidArgument	if the size of provided wrapped key is unsupported.
CryptoLogicError	if the context was not initialized by a key value.

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

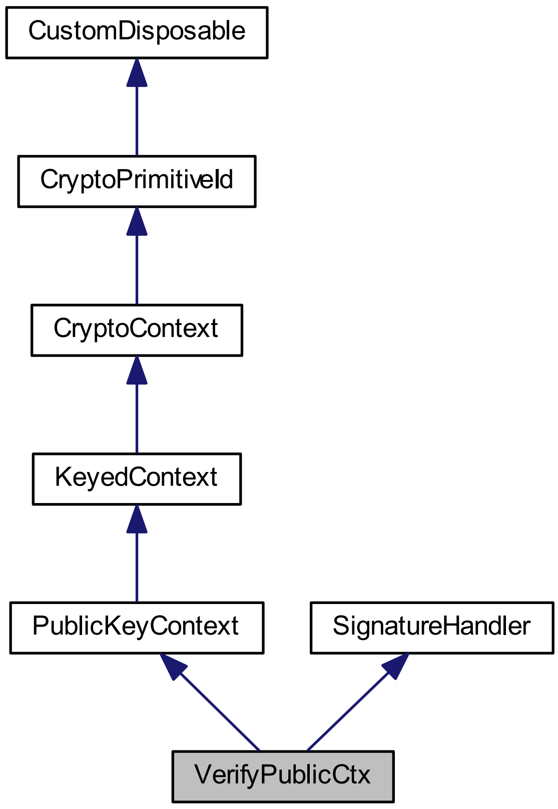
⌋(*RS_CRYPTO_02404*, *RS_CRYPTO_02115*)

9.64 VerifyPublicCtx Interface Reference

Inheritance diagram for VerifyPublicCtx:



Collaboration diagram for VerifyPublicCtx:



Public Types

- typedef std::unique_ptr< VerifyPublicCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual bool verify (const HashFunctionCtx &hash, const Signature &signature) const noexcept(false)=0
- virtual bool verify (ReadOnlyMemRegion hash, ReadOnlyMemRegion signature) const noexcept(false)=0

Additional Inherited Members

9.64.1 Detailed Description

[SWS_CRYPTO_24100] Interface definition: ⌈ Signature Verification Public key Context interface

⌋*(RS_CRYPTO_02204)*

9.64.2 Member Typedef Documentation

9.64.2.1 typedef std::unique_ptr<VerifyPublicCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_24101] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.64.3 Member Function Documentation

9.64.3.1 virtual bool verify (const HashFunctionCtx & *hash*, const Signature & *signature*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24111] Interface definition: ⌈ Verify signature by a digest value stored in the hash-function context.

Parameters

`in`	*hash*	A finalized hash-function context that contains a digest value ready for the verification.
`in`	*signature*	A signature object for the verification.

Returns

`true` if the signature was verified successfully and `false` otherwise.

Note

If any of the 4 mentioned below conditions was violated then this method returns `false`:

- This method must control compliance of the real hash-function algorithm to the hash algorithm specified in the signature!
- This method must check equality of the real public key COUID and corresponded dependency COUID in the signature!
- This method must control the hash-function algorithm ID compliance to the signature algorithm specification of the context!
- This method must control compliance of the signature algorithm ID in the signature object and the signature verification context!

Exceptions

CryptoLogicError	if the context was not initialized by a key value or the method `hash.finish()` was not called before this method call.

⌋(*RS_CRYPTO_02311*)

9.64.3.2 virtual bool verify (ReadOnlyMemRegion *hash*, ReadOnlyMemRegion *signature*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24112] Interface definition: ⌈ Verify signature by a directly provided hash value.

Parameters

`in`	*hash*	A hash value for the verification.
`in`	*signature*	A signature value for the verification.

Returns

`true` if the signature was verified successfully and `false` otherwise.

Note

If sizes of the hash or signature values are incorrect then this method returns false (before starting of any calculations)!

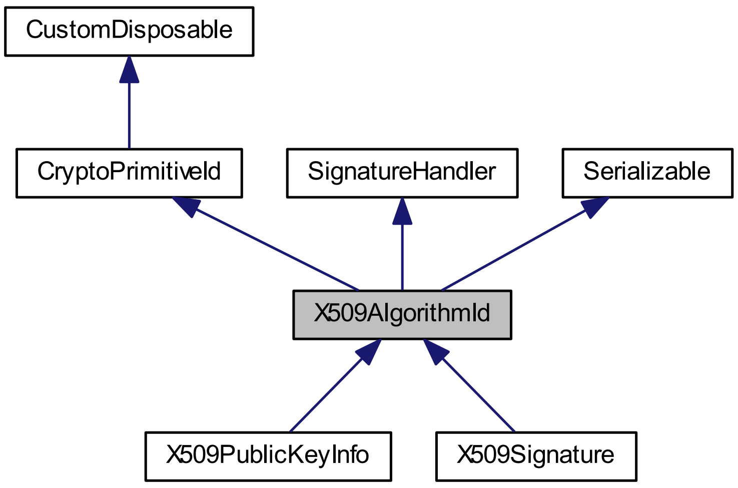
Exceptions

CryptoLogicError	if the context was not initialized by a key value.

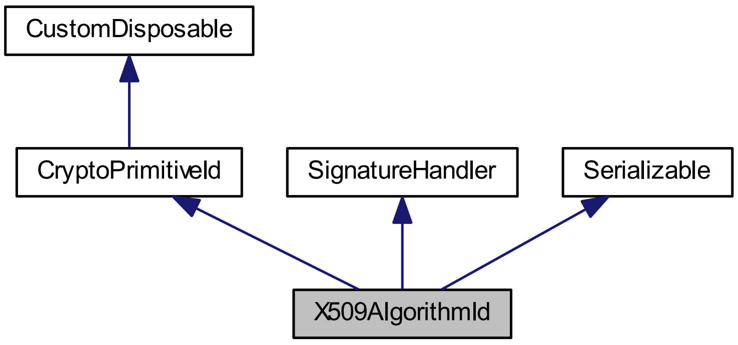
⌋(*RS_CRYPTO_02311*)

9.65 X509AlgorithmId Interface Reference

Inheritance diagram for X509AlgorithmId:



Collaboration diagram for X509AlgorithmId:



Public Types

- typedef std::unique_ptr< const X509AlgorithmId, CustomDeleter > uptrc_t

Public Member Functions

- virtual bool hasDomainParameters () const noexcept(true)=0
- virtual DomainParameters::sptrc_t getDomainParameters (ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual bool isSameParameters (const DomainParameters &params) const noexcept(true)=0

Additional Inherited Members

9.65.1 Detailed Description

[SWS_CRYPTO_24200] Interface definition: ⌈ X.509 Algorithm ID interface.

Note

> If instance of this interface is created for unsupported algorithm, then method `getPrimitiveId()` will return `ALGID_UNKNOWN!`

⌋*(RS_CRYPTO_02307)*

9.65.2 Member Typedef Documentation

9.65.2.1 typedef std::unique_ptr<const X509AlgorithmId, CustomDeleter> uptrc_t

[SWS_CRYPTO_24201] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.65.3 Member Function Documentation

9.65.3.1 virtual bool hasDomainParameters ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24211] Interface definition: ⌈ Verifies presence of domain parameters in this object.

Returns

> `true` if this instance includes domain parameters.

⌋(*RS_CRYPTO_02108*)

9.65.3.2 virtual DomainParameters::sptrc_t getDomainParameters (ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_24212] Interface definition: ⌈ Get domain parameters object associated with the public key of the subject.

Parameters

`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

> Smart unique pointer to the created Domain parameters object associated with the public key of the subject.

Note

> This method returns `nullptr` if this instance does not include domain parameters.

Exceptions

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02306*, *RS_CRYPTO_02404*)

9.65.3.3 virtual bool isSameParameters (const DomainParameters & *params*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_24213] Interface definition: ⌈ Verifies the sameness of the provided and internally stored domain parameters.

Parameters

in	*params*	A domain parameters object for comparison.

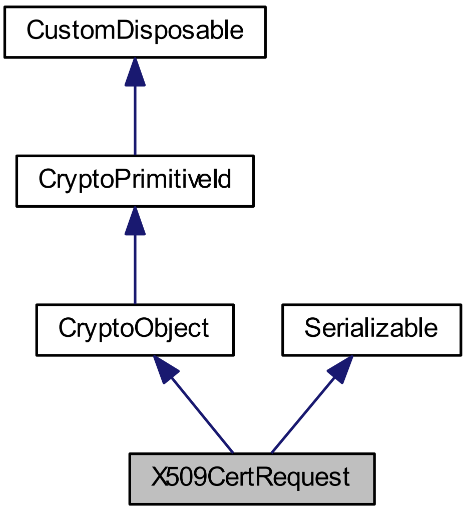
Returns

`true` if values of the stored domain parameters and object provided by the argument are identical.

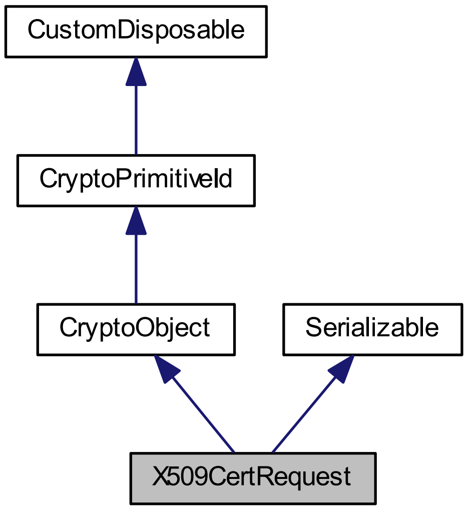
⌋(*RS_CRYPTO_02108*)

9.66 X509CertRequest Interface Reference

Inheritance diagram for X509CertRequest:



Collaboration diagram for X509CertRequest:



Public Types

- typedef std::unique_ptr< const X509CertRequest > uptrc_t

Public Member Functions

- virtual bool verify (HashFunctionCtx &hash, VerifyPublicCtx &verifier) const noexcept(false)=0
- virtual unsigned version () const noexcept(true)=0
- virtual const X509Signature & signature () const noexcept(true)=0
- virtual const X509PublicKeyInfo & subjectPublicKeyInfo () const noexcept(true)=0

Additional Inherited Members

9.66.1 Detailed Description

[SWS_CRYPTO_24300] Interface definition: ⌈ Certificate Request object interface
⌋*(RS_CRYPTO_02307)*

9.66.2 Member Typedef Documentation

9.66.2.1 typedef std::unique_ptr<const X509CertRequest> uptrc_t

[SWS_CRYPTO_24301] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.66.3 Member Function Documentation

9.66.3.1 virtual bool verify (HashFunctionCtx & *hash*, VerifyPublicCtx & *verifier*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24311] Interface definition: ⌈ Verifies self-signed signature of the certificate request.

Parameters

`in`	*hash*	Temporary hash-function context that should be used in the call (the Alg ID can be got as `X509Signature::getPrimitiveId()`).
`in`	*verifier*	Temporary signature verification context that should be used in the call (the Alg ID can be got as `X509Signature::requiredHashAlgId()`).

Returns

true if the signature is correct.

Note

This method uses key values and domain parameters stored inside the object!

Exceptions

CryptoInvalidArgument	if the hash or verifier arguments are configured for algorithms different from used for this request signature.

⌋*(RS_CRYPTO_02306)*

9.66.3.2 virtual unsigned version ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24312] Interface definition: ⌈ Returns format version of the certificate request.

Returns

Format version of the certificate request.

⌋(*RS_CRYPTO_02311*)

9.66.3.3 virtual const X509Signature& signature ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24313] Interface definition: ⌈ Returns signature object of the request.

Returns

Signature object of the request.

⌋(*RS_CRYPTO_02306*)

9.66.3.4 virtual const X509PublicKeyInfo& subjectPublicKeyInfo ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24314] Interface definition: ⌈ Returns subject public key included to the request.

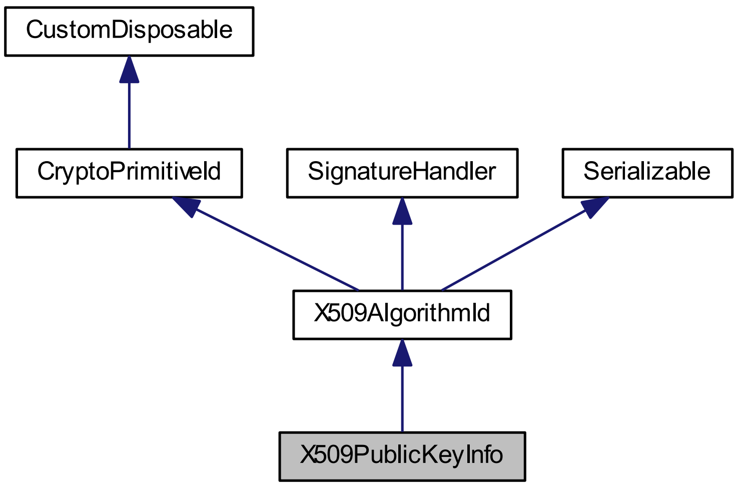
Returns

Subject public key included to the request.

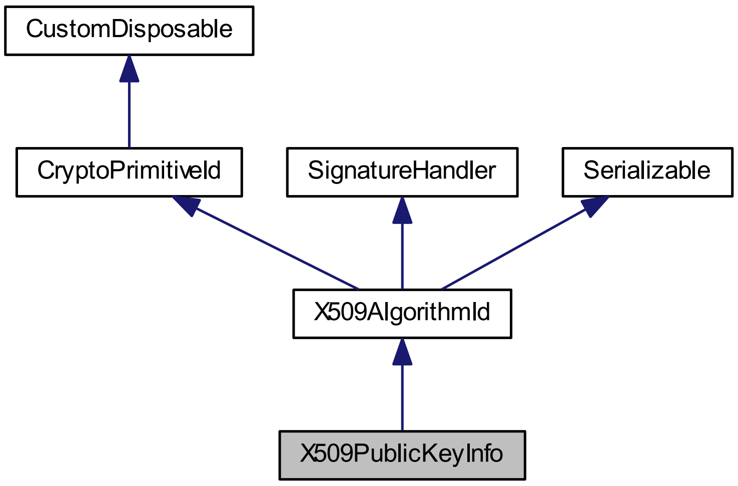
⌋(*RS_CRYPTO_02306*)

9.67 X509PublicKeyInfo Interface Reference

Inheritance diagram for X509PublicKeyInfo:



Collaboration diagram for X509PublicKeyInfo:



Public Types

- typedef std::unique_ptr< const X509PublicKeyInfo, CustomDeleter > uptrc_t

Public Member Functions

- virtual PublicKey::uptrc_t getPublicKey (DomainParameters::sptrc_t params=nullptr, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0
- virtual bool isSameKey (const PublicKey &publicKey) const noexcept(true)=0

Additional Inherited Members

9.67.1 Detailed Description

[SWS_CRYPTO_24400] Interface definition: ⌈ X.509 Public Key Information interface.

⌋*(RS_CRYPTO_02307)*

9.67.2 Member Typedef Documentation

9.67.2.1 typedef std::unique_ptr<const X509PublicKeyInfo, CustomDeleter> uptrc_t

[SWS_CRYPTO_24401] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.67.3 Member Function Documentation

9.67.3.1 virtual PublicKey::uptrc_t getPublicKey (DomainParameters::sptrc_t *params* = `nullptr`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24411] Interface definition: ⌈ Get public key object of the subject.

Parameters

`in`	*params*	An optional pointer to Domain Parameters required for full specification of the transformation.

in	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Smart unique pointer to the created public key of the subject.

Exceptions

CryptoInvalidArgument	if the argument params refers to inappropriate domain parameters.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

⌋(*RS_CRYPTO_02108*, *RS_CRYPTO_02306*, *RS_CRYPTO_02404*)

9.67.3.2 virtual bool isSameKey (const PublicKey & *publicKey*) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_24412] Interface definition: ⌈ Verifies the sameness of the provided and kept public keys.

Parameters

in	*publicKey*	A public key object for comparison.

Returns

true if values of the stored public key and object provided by argument are identical.

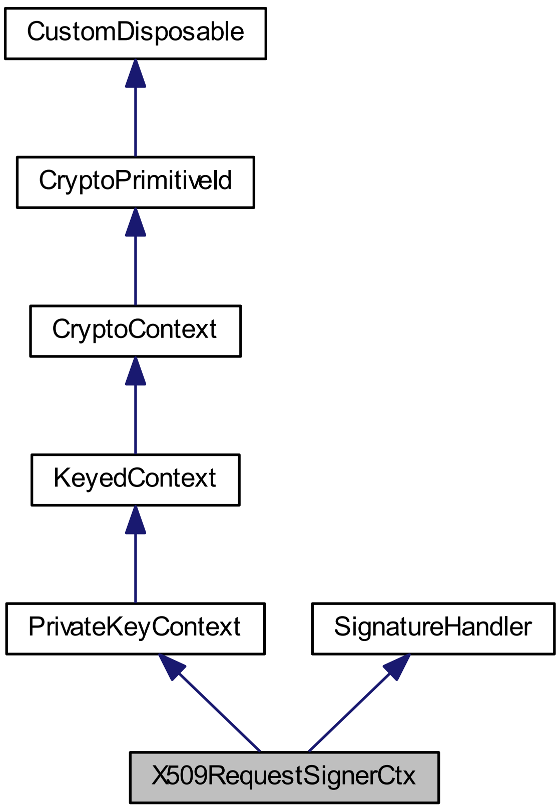
Note

This method compare the public key values only.

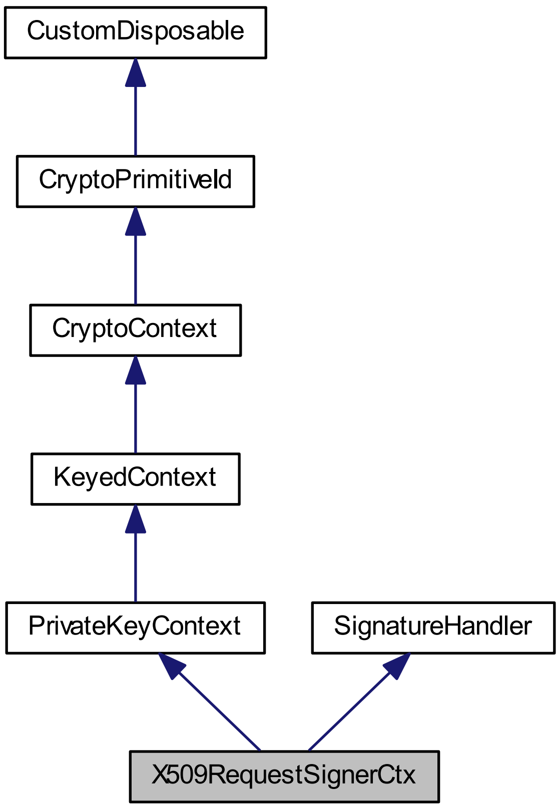
⌋(*RS_CRYPTO_02311*)

9.68 X509RequestSignerCtx Interface Reference

Inheritance diagram for X509RequestSignerCtx:



Collaboration diagram for X509RequestSignerCtx:



Public Types

- typedef std::unique_ptr< X509RequestSignerCtx, CustomDeleter > uptr_t

Public Member Functions

- virtual X509CertRequest::uptrc_t createCertRequest (ReadOnlyMemRegion derSubjectDN, ReadOnlyMemRegion ∗x509Extensions=nullptr, unsigned version=1, ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.68.1 Detailed Description

[SWS_CRYPTO_24500] Interface definition: ⌈ X.509 Request Signer Context interface

Note

> Any private key (including keys without `Key::ALLOW_SIGNATURE` attribute) can be loaded to this interface context!

⌋*(RS_CRYPTO_02307)*

9.68.2 Member Typedef Documentation

9.68.2.1 typedef std::unique_ptr<X509RequestSignerCtx, CustomDeleter> uptr_t

[SWS_CRYPTO_24501] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.68.3 Member Function Documentation

9.68.3.1 virtual X509CertRequest::uptrc_t createCertRequest (ReadOnlyMemRegion *derSubjectDN*, ReadOnlyMemRegion ∗ *x509Extensions* = `nullptr`, unsigned *version* = `1`, ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24511] Interface definition: ⌈ Creates certification request for a private key loaded to the context.

Parameters

`in`	*derSubjectDN*	A DER-encoded subject distinguished name (DN) of the private key owner.
`in`	*x509Extensions*	A DER-encoded X.509 Extensions that should be included to the certification request.
`in`	*version*	Format version of the target certification request.
`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Unique smart pointer to created certification request.

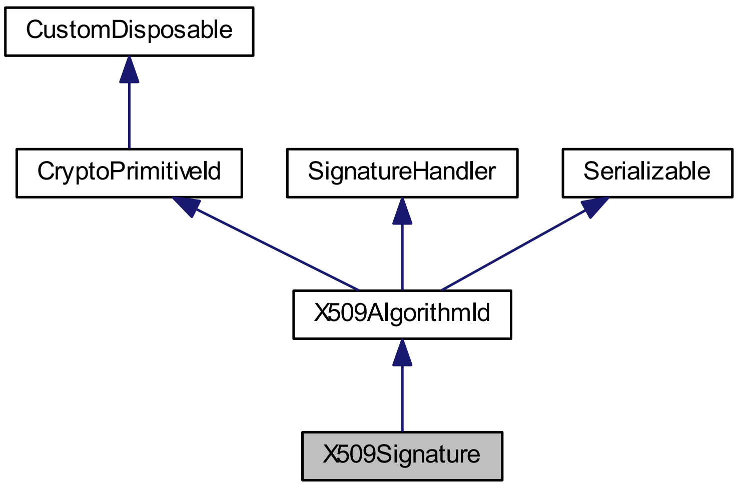
Exceptions

CryptoInvalidArgument	if any of arguments has incorrect/unsupported value.
CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

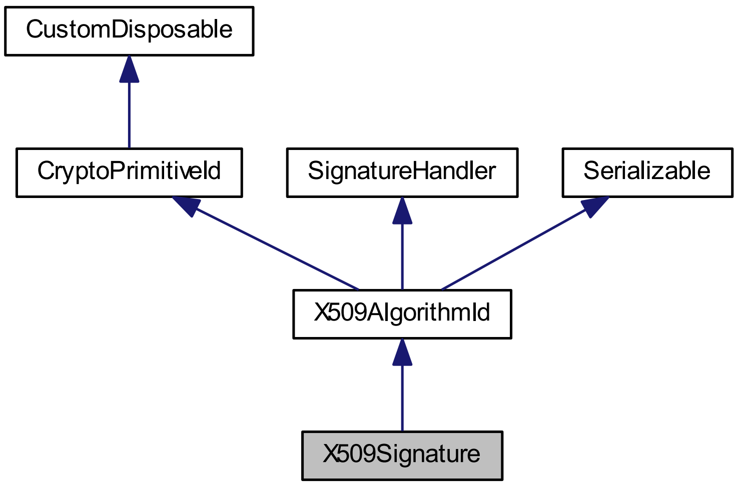
⌋*(RS_CRYPTO_02306, RS_CRYPTO_02404)*

9.69 X509Signature Interface Reference

Inheritance diagram for X509Signature:



Collaboration diagram for X509Signature:



Public Types

- typedef std::unique_ptr< const X509Signature, CustomDeleter > uptrc_t

Public Member Functions

- virtual Signature::uptrc_t getSignature (ReservedObjectIndex_t reservedIndex=ALLOC_OBJECT_ON_HEAP) const noexcept(false)=0

Additional Inherited Members

9.69.1 Detailed Description

[SWS_CRYPTO_24600] Interface definition: ⌈ X.509 Signature interface (X.509 Algorithm Identifier & Signature Value).

⌋*(RS_CRYPTO_02307)*

9.69.2 Member Typedef Documentation

9.69.2.1 typedef std::unique_ptr<const X509Signature, CustomDeleter> uptrc_t

[SWS_CRYPTO_24601] Interface definition: ⌈ Unique smart pointer of the interface.

⌋ *(RS_CRYPTO_02404)*

9.69.3 Member Function Documentation

9.69.3.1 virtual Signature::uptrc_t getSignature (ReservedObjectIndex_t *reservedIndex* = ALLOC_OBJECT_ON_HEAP) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_24611] Interface definition: ⌈ Get signature object of the hosted DER-straucture.

Parameters

`in`	*reservedIndex*	An optional index of reserved Object slot that should be used for this allocation or default marker, which says to allocate on the heap.

Returns

Smart unique pointer to the created signature object.

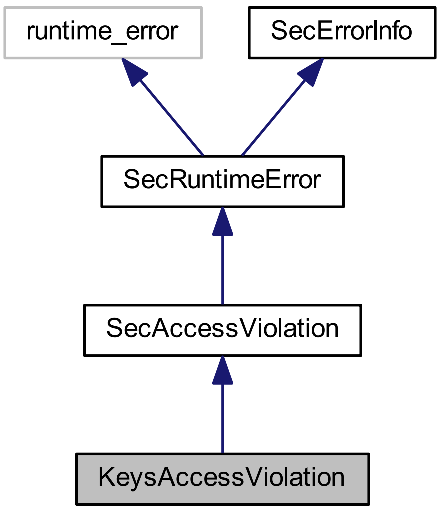
Exceptions

CryptoBadAlloc	if the slot specified by `reservedIndex` is busy yet or it was not allocated or its size is not enough for placing of the target object or if `(reservedIndex == ALLOC_OBJECT_ON_HEAP)`, but allocation memory on the heap is impossible.

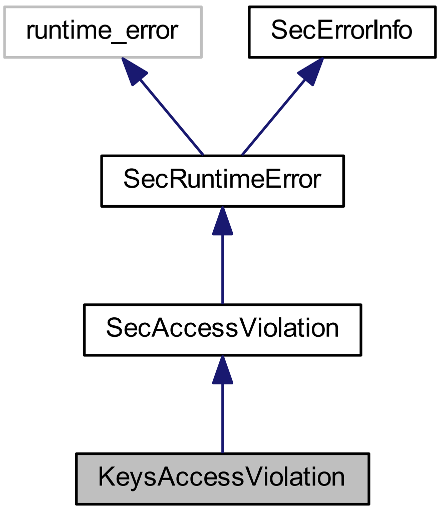
⌋ *(RS_CRYPTO_02306, RS_CRYPTO_02404)*

9.70 KeysAccessViolation Interface Reference

Inheritance diagram for KeysAccessViolation:



Collaboration diagram for KeysAccessViolation:



Additional Inherited Members

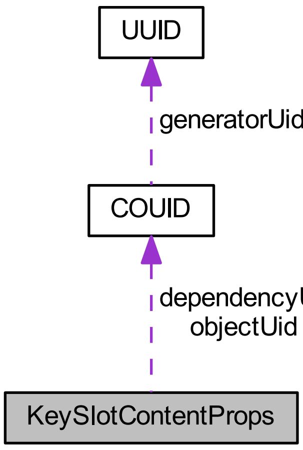
9.70.1 Detailed Description

[SWS_CRYPTO_39900] Interface definition: ⌈ An interface of the Key Storage Access Violation exception.

⌋*(RS_CRYPTO_02310)*

9.71 KeySlotContentProps Struct Reference

Collaboration diagram for KeySlotContentProps:



9.71.1 Detailed Description

[SWS_CRYPTO_30004] Interface definition: ⌈ Properties of current Key Slot Content, i.e. of a current instance stored to the Key Slot.

Note

A value of the allowedUsage field is bitwise AND of the common usage flags defined at run-time and the usage flags defined by the UserPermissions prototype for current Actor (application).

⌋*(RS_CRYPTO_02005, RS_CRYPTO_02404)*

9.71.2 Member Data Documentation

9.71.2.1 COUID objectUid

UID of a Crypto Object stored to the slot.

9.71.2.2 COUID dependencyUid

UID of a Crypto Object from which depends object stored to the slot.

9.71.2.3 CryptoAlgId_t algId

Cryptoalgorithm of actual object stored to slot.

9.71.2.4 std::size_t objectSize

Actual size of an object currently stored to the slot.

9.71.2.5 AllowedUsageFlags_t allowedUsage

Actual usage restriction flags of an object stored to the slot for the current application.

9.71.2.6 CryptoObjectType objectType

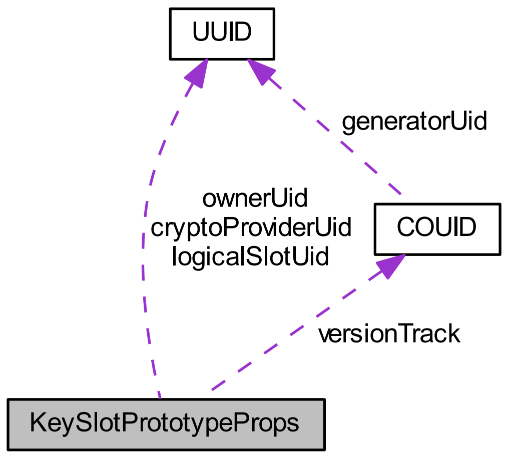
Actual type of an object stored to the slot.

9.71.2.7 bool exportability

Effective value of the exportability attribute of current object.

9.72 KeySlotPrototypeProps Struct Reference

Collaboration diagram for KeySlotPrototypeProps:



9.72.1 Detailed Description

[SWS_CRYPTO_30002] Interface definition: ⌈ Prototyped Properties of a Key Slot.

⌋*(RS_CRYPTO_02009, RS_CRYPTO_02110, RS_CRYPTO_02116, RS_CRYPTO_02404)*

9.72.2 Member Data Documentation

9.72.2.1 LogicalSlotUID_t logicalSlotUid

Logical Slot UID defined at Design phase of the Owner software.

9.72.2.2 ActorUID_t ownerUid

UID of an Owner actor (application) of this key slot.

9.72.2.3 CryptoProviderUID_t cryptoProviderUid

UUID of a Crypto Provider assigned for this key slot processing (at Integration stage).

9.72.2.4 COUID versionTrack

COUID of last object stored to this slot (versionTrack.generatorUid restricts objects' source).

9.72.2.5 CryptoAlgId_t algId

Cryptoalgorithm restriction (`ALGID_ANY` means without restriction). Algorithm can be partially specified: family & length, mode, padding.

9.72.2.6 std::size_t slotCapacity

Capacity of the slot in bytes.

9.72.2.7 CryptoObjectType objectType

Restriction of an object type that can be stored the slot (if `CryptoObjectType::UNKNOWN` then without restriction).

9.72.2.8 CryptoObjectType dependecyType

Type of object from which depends type prototyped for this slot (if `CryptoObjectType::NONE` then without dependency).

9.72.2.9 VersionControlType versionControl

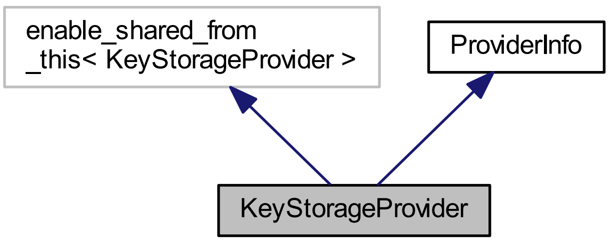
Version Control Type selects a rule restriction source of stored objects.

9.72.2.10 bool exportability

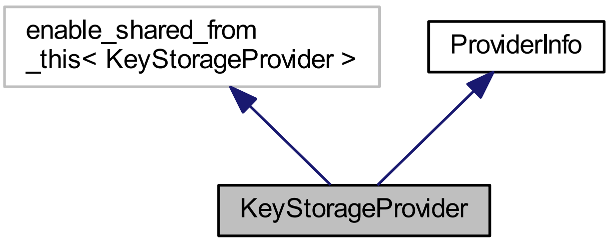
Exportability restriction: if false then any exportable object cannot be saved to this slot.

9.73 KeyStorageProvider Interface Reference

Inheritance diagram for KeyStorageProvider:



Collaboration diagram for KeyStorageProvider:



Public Types

- typedef std::shared_ptr< KeyStorageProvider > sptr_t
- using SlotNumber_t = std::size_t
- using ObjectUid_t = COUID
- using SlotUid_t = LogicalSlotUID_t
- using ContentType_t = CryptoObjectType

Public Member Functions

- virtual ∼KeyStorageProvider ()=default

- virtual SlotNumber_t findSlot (const SlotUid_t &slotUid, CryptoProviderUID_t &providerUid, SlotNumber_t previousFound=INVALID_SLOT) const noexcept(true)=0
- virtual SlotNumber_t findObject (const ObjectUid_t &objectUid, ContentType_t objectType, CryptoProviderUID_t &providerUid, SlotNumber_t previousFound=INVALID_SLOT) const noexcept(true)=0
- virtual bool isEmpty (SlotNumber_t slotNum) const noexcept(false)=0
- virtual TrustedContainer::uptrc_t openExisting (SlotNumber_t slotNum) noexcept(false)=0
- virtual TrustedContainer::uptr_t openEmpty (SlotNumber_t slotNum) noexcept(false)=0
- virtual void saveVolatile (SlotNumber_t slotNum, const TrustedContainer &container) noexcept(false)=0
- virtual void clear (SlotNumber_t slotNum) noexcept(false)=0
- virtual void getPrototypedProps (SlotNumber_t slotNum, KeySlotPrototypeProps &props) noexcept(false)=0
- virtual void getContentProps (SlotNumber_t slotNum, KeySlotContentProps &props) noexcept(false)=0
- virtual void getDefaultCryptoProviderUid (SlotNumber_t slotNum, CryptoProviderUID_t &providerUid) noexcept(false)=0
- virtual void getOwner (SlotNumber_t slotNum, ActorUID_t &ownerUid) noexcept(false)=0
- virtual std::size_t getUsers (SlotNumber_t slotNum, std::vector< UserPermissions > *users=nullptr) noexcept(false)=0

Static Public Attributes

- static const SlotNumber_t INVALID_SLOT = static_cast<SlotNumber_t>(-1LL)

Additional Inherited Members

9.73.1 Detailed Description

[SWS_CRYPTO_30100] Interface definition: ⌈ Key Storage Provider interface.

Note

> Any object is uniquely identified by the combination of its GUID and type.
> Only a single instance of any object is allowed in scope of each Crypto Provider in the persistent storage!
> HSMs/TPMs implementing non-extractable keys should use own copies of externally supplied crypto objects.
> A few software Crypto Providers can share single key slot if they support same format.

⌋*(RS_CRYPTO_02109, RS_CRYPTO_02305, RS_CRYPTO_02401)*

9.73.2 Member Typedef Documentation

9.73.2.1 typedef std::shared_ptr<KeyStorageProvider> sptr_t

[SWS_CRYPTO_30101] Interface definition: ⌈ Shared smart pointer of the interface.

⌋*(RS_CRYPTO_02311)*

9.73.2.2 using SlotNumber_t = std::size_t

[SWS_CRYPTO_30102] Interface definition: ⌈ The Slot Number is intended for direct addressing of the "Key Slots".

Note

> Each "Key Slot" is intended for storing of a single cryptographic object.
> "Actor" is a permanently identifiable process defined by the Startup Configuration of an Executable.
> Access control management is based on the key slot attributes: (1) Each persistent slot always has only one "Owner" Actor that can save an object to the slot or clear it. Actor is responsible for the slot content. Only "Owner" Actor can execute export operation of a crypto object (if it is allowed by the object attributes). (2) A "User" access right for a slot can be granted to an Actor application by the Owner's manifest. The "User" access means the right to load a crypto object from the slot to the Crypto Provider via the trusted container interface. Additionally all "User" Actors obtain the "Exportability" attribute enforced to "false" (i.e. they cannot export objects independently from the actual attribute value).

⌋*(RS_CRYPTO_02404)*

9.73.2.3 using ObjectUid_t = COUID

[SWS_CRYPTO_30104] Interface definition: ⌈ Definition of an object UID type.

⌋(RS_CRYPTO_02005)

9.73.2.4 using SlotUid_t = LogicalSlotUID_t

[SWS_CRYPTO_30105] Interface definition: ⌈ Definition of an object UID type.

⌋(RS_CRYPTO_02404, RS_CRYPTO_02405)

9.73.2.5 using ContentType_t = CryptoObjectType

[SWS_CRYPTO_30106] Interface definition: ⌈ Definition of a slot content type.

⌋(RS_CRYPTO_02311)

9.73.3 Constructor & Destructor Documentation

9.73.3.1 virtual ∼KeyStorageProvider ( ) `[virtual]`, `[default]`

[SWS_CRYPTO_30110] Interface definition: ⌈ Destructor.

⌋(RS_CRYPTO_02311)

9.73.4 Member Function Documentation

9.73.4.1 virtual SlotNumber_t findSlot (const SlotUid_t & *slotUid,* CryptoProviderUID_t & *providerUid,* SlotNumber_t *previousFound* = INVALID_SLOT) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30111] Interface definition: ⌈ Finds a slot number by the Logic (persistent) Slot UID.

Parameters

`in`	*slotUid*	Logic Slot UID.
`in,out`	*providerUid*	A UID of Crypto Provider responsible for servicing of the slot. A non-zero input value restricts a search scope to specific Crypto Provider.
`in`	*previousFound*	A number of previous found key slot (the search will start from next slot number). `INVALID_SLOT` means start the search from the begin.

Returns

Number of found slot or `INVALID_SLOT` if a slot with such logic UID was not found.

Note

If the provider UID has the zero value then the search is executed through whole Key Storage (without limitation to any specific Crypto Provider).
If an application needs to find all instances of a logical key slot (through all Crypto Providers) then this method should be called multiple times until it will return `INVALID_SLOT`.

⌋(*RS_CRYPTO_02405*)

9.73.4.2 virtual SlotNumber_t findObject (const ObjectUid_t & *objectUid,* ContentType_t *objectType,* CryptoProviderUID_t & *providerUid,* SlotNumber_t *previousFound* = INVALID_SLOT) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30112] Interface definition: ⌈ Finds a slot number by the Crypto Object's UID and type.

Parameters

`in`	*objectUid*	Target object UID.
`in`	*objectType*	Type of the target object.
`in,out`	*providerUid*	A UID of Crypto Provider responsible for servicing of the slot. A non-zero input value restricts a search scope to specific Crypto Provider.
`in`	*previousFound*	A number of previous found key slot (the search will start from next slot number). INVALID_SLOT means start the search from the begin.

Returns

Number of a slot containing the found object or `INVALID_SLOT` if the object was not found.

Note

> If the provider UID has the zero value then the search is executed through whole Key Storage (without limitation to any specific Crypto Provider).
> If an application needs to find all instances of a crypto objects (through all Crypto Providers) then this method should be called multiple times until it will return `INVALID_SLOT`.

⌋(*RS_CRYPTO_02004*, *RS_CRYPTO_02005*)

9.73.4.3 virtual bool isEmpty (SlotNumber_t *slotNum*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30113] Interface definition: ⌈ Checks the slot for emptiness.

Parameters

`in`	*slotNum*	Target slot number.

Returns

> "true" if the slot is empty or "false" otherwise.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which has no any ("Owner" or "User") access rights to the key slot.

⌋(*RS_CRYPTO_02311*)

9.73.4.4 virtual TrustedContainer::uptrc_t openExisting (SlotNumber_t *slotNum*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30114] Interface definition: ⌈ Opens a slot containing an existing object and associate a trusted container to it (for reading only).

Parameters

`in`	*slotNum*	Target slot number.

Returns

Unique smart pointer to allocated trusted container associated with the slot content.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which has no any ("Owner" or "User") access rights to the key slot.

⌋(*RS_CRYPTO_02311*)

9.73.4.5 virtual TrustedContainer::uptr_t openEmpty (SlotNumber_t *slotNum*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30115] Interface definition: ⌈ Associates a trusted container with an empty slot for following writting firstly and then for reading only.

Parameters

`in`	*slotNum*	Target slot number.

Returns

Unique smart pointer to allocated trusted container associated to the slot's space.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which is not "Owner" of the key slot.

⌋(*RS_CRYPTO_02311*)

9.73.4.6 virtual void saveVolatile (SlotNumber_t *slotNum*, const TrustedContainer & *container*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30116] Interface definition: ⌈ Saves a content of a temporary (volatile) trusted container to persistent slot on the storage.

Parameters

in	*slotNum*	Target slot number. The slot should be empty!
in	*container*	Source volatile container.

Exceptions

SecRuntimeError	if any of mentioned below conditions is satisfied: • the target slot is not empty; • an object with same COUID and type already exists in the storage (or equally if the source container is not volatile); • an object in the container is "session" or doesn't satisfy the slot restrictions (including version control); • the slot number is incorrect (the slot is not allocated); • the source container is volatile.
KeysAccessViolation	if this method is called by an Actor which is not "Owner" of the key slot.

⌋(*RS_CRYPTO_02311*)

9.73.4.7 virtual void clear (SlotNumber_t *slotNum*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30117] Interface definition: ⌈ Clears the slot identified by its number.

Parameters

in	*slotNum*	Target slot number.

Note

> This method must perform a secure cleanup without the ability to restore the object data!

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which is not "Owner" of the key slot.

⌋(*RS_CRYPTO_02009*)

9.73.4.8 virtual void getPrototypedProps (SlotNumber_t *slotNum*, KeySlotPrototypeProps & *props*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30118] Interface definition: ⌈ Gets the prototyped properties of the key slot.

Parameters

`in`	*slotNum*	Target slot number.
`out`	*props*	Output buffer for storing the prototype properties of the key slot.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which has no any ("Owner" or "User") access rights to the key slot.

⌋(*RS_CRYPTO_02110*)

9.73.4.9 virtual void getContentProps (SlotNumber_t *slotNum*, KeySlotContentProps & *props*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30119] Interface definition: ⌈ Gets an actual properties of a content in the key slot..

Parameters

`in`	*slotNum*	Target slot number.
`out`	*props*	Output buffer for storing an actual properties of a content in the key slot.

Note

If this method called by a User Actor then always: `props.exportability == false`.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).
KeysAccessViolation	if this method is called by an Actor which has no any ("Owner" or "User") access rights to the key slot.

⌋(*RS_CRYPTO_02311*)

9.73.4.10 virtual void getDefaultCryptoProviderUid (SlotNumber_t *slotNum*, CryptoProviderUID_t & *providerUid*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_30120] Interface definition: ⌈ Gets UID of an Actor granted by the Owner rights for the key slot.

Parameters

`in`	*slotNum*	Target slot number.
`out`	*providerUid*	A UID of Crypto Provider responsible for servicing of the slot.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).

⌋(*RS_CRYPTO_02401*)

9.73.4.11 virtual void getOwner (SlotNumber_t *slotNum*, ActorUID_t & *ownerUid*) `[pure virtual], [noexcept]`

[SWS_CRYPTO_30121] Interface definition: ⌈ Gets UID of an Actor granted by the Owner rights for the key slot.

Parameters

`in`	*slotNum*	Target slot number.

out	*ownerUid*	Output buffer for storing the Owner UID of the key slot.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated).

⌋(*RS_CRYPTO_02009*)

9.73.4.12 virtual std::size_t getUsers (SlotNumber_t *slotNum*, std::vector< UserPermissions > * *users* = `nullptr`) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_30122] Interface definition: ⌈ Gets Users' Permissions list of all Actors granted by the User rights for the key slot.

Parameters

in	*slotNum*	Target slot number.
out	*users*	Optional pointer to a vector for storing output list.

Returns

Number of Users in the output list.

Note

If `(users != nullptr)` then capacity of the output vector should be enough for storing permissions of all Users.

Exceptions

SecRuntimeError	if the slot number is incorrect (the slot is not allocated) or capacity of the output vector is not enough.

⌋(*RS_CRYPTO_02009*, *RS_CRYPTO_02114*)

9.73.5 Member Data Documentation

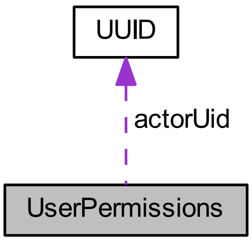
9.73.5.1 const SlotNumber_t INVALID_SLOT = static_cast<SlotNumber_t>(-1LL) `[static]`

32 or 64 bit depending from the system. **[SWS_CRYPTO_30103] Interface definition:** ⌈ A reserved slot number that cannot be used for addressing of real slots of the storage.

⌋*(RS_CRYPTO_02311)*

9.74 UserPermissions Struct Reference

Collaboration diagram for UserPermissions:



9.74.1 Detailed Description

[SWS_CRYPTO_30003] Interface definition: ⌈ Key slot User's Permissions prototype defined at the Design (or the Integration) stage.

⌋*(RS_CRYPTO_02009, RS_CRYPTO_02114, RS_CRYPTO_02404)*

9.74.2 Member Data Documentation

9.74.2.1 ActorUID_t actorUid

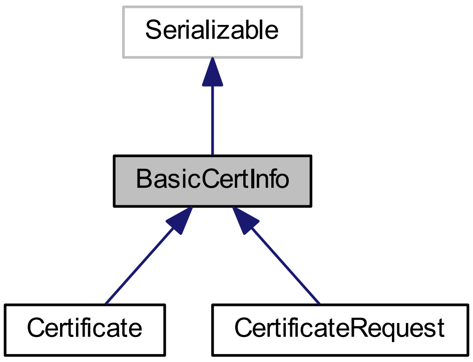
UID of a User Actor (application). User actor has right to load the crypto object to suitable crypto contexts. ∗/.

9.74.2.2 AllowedUsageFlags_t allowedUsage

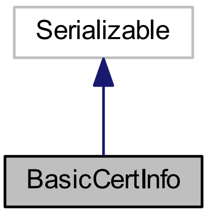
Restriction flags of allowed usage of a key stored to the slot for application defined by the actorUid. ∗/.

9.75 BasicCertInfo Interface Reference

Inheritance diagram for BasicCertInfo:



Collaboration diagram for BasicCertInfo:



Public Types

- typedef std::uint32_t KeyConstraints_t

Public Member Functions

- virtual const crypto::X509PublicKeyInfo & subjectPubKey () const noexcept(true)=0
- virtual const X509DN & subjectDN () const noexcept(true)=0
- virtual bool isCA () const noexcept(true)=0
- virtual std::uint32_t pathLimit () const noexcept(true)=0
- virtual KeyConstraints_t constraints () const noexcept(true)=0

9.75.1 Detailed Description

[SWS_CRYPTO_40100] Interface definition: ⌈ Basic Certificate Information interface.

⌋*(RS_CRYPTO_02306)*

9.75.2 Member Typedef Documentation

9.75.2.1 typedef std::uint32_t KeyConstraints_t

[SWS_CRYPTO_40101] Interface definition: ⌈ X.509 v3 Key Constraints type definition.

⌋*(RS_CRYPTO_02311)*

9.75.3 Member Function Documentation

9.75.3.1 virtual const crypto::X509PublicKeyInfo& subjectPubKey ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40111] Interface definition: ⌈ Get the subject public key interface.

Returns

Constant reference of the subject public key interface.

⌋*(RS_CRYPTO_02311)*

9.75.3.2 virtual const X509DN& subjectDN ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40112] Interface definition: ⌈ Get the subject DN.

Returns

subject DN

⌋(*RS_CRYPTO_02311*)

9.75.3.3 virtual bool isCA ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_40113] Interface definition: ⌈ Find out whether this is a CA request.

Returns

true if it is a CA request, false otherwise.

⌋(*RS_CRYPTO_02311*)

9.75.3.4 virtual std::uint32_t pathLimit ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_40114] Interface definition: ⌈ Return the constraint on the path length defined in the BasicConstraints extension.

Returns

path limit

⌋(*RS_CRYPTO_02311*)

9.75.3.5 virtual KeyConstraints_t constraints ( ) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_40115] Interface definition: ⌈ Get the key constraints for the key associated with this PKCS#10 object.

Returns

key constraints

⌋(*RS_CRYPTO_02311*)

9.75.4 Member Data Documentation

9.75.4.1 const KeyConstraints_t CONSTR_NONE = 0 `[static]`

No constraints.

9.75.4.2 const KeyConstraints_t CONSTR_DIGITAL_SIGNATURE = 0x08000 `[static]`

The key can be used for digital signature production.

9.75.4.3 const KeyConstraints_t CONSTR_NON_REPUDIATION = 0x04000 `[static]`

The key can be used in cases requiring the "non-repudiation" guarantee.

9.75.4.4 const KeyConstraints_t CONSTR_KEY_ENCIPHERMENT = 0x02000 `[static]`

The key can be used for key encipherment.

9.75.4.5 const KeyConstraints_t CONSTR_DATA_ENCIPHERMENT = 0x01000 `[static]`

The key can be used for data encipherment.

9.75.4.6 const KeyConstraints_t CONSTR_KEY_AGREEMENT = 0x00800 `[static]`

The key can be used for a key agreement protocol execution.

9.75.4.7 const KeyConstraints_t CONSTR_KEY_CERT_SIGN = 0x00400 `[static]`

The key can be used for certificates signing.

9.75.4.8 const KeyConstraints_t CONSTR_CRL_SIGN = 0x00200 `[static]`

The key can be used for Certificates Revokation Lists (CRL) signing.

9.75.4.9 const KeyConstraints_t CONSTR_ENCIPHER_ONLY = 0x00100 `[static]`

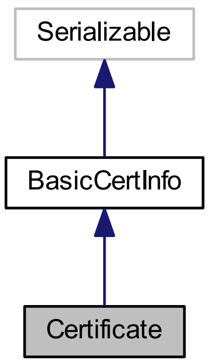
The enciphermet key can be used for enciphering only.

9.75.4.10 const KeyConstraints_t CONSTR_DECIPHER_ONLY = 0x00080 `[static]`

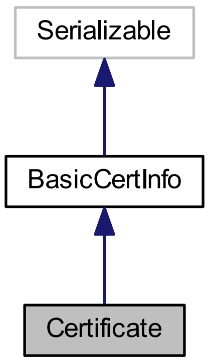
The enciphermet key can be used for deciphering only.

9.76 Certificate Interface Reference

Inheritance diagram for Certificate:



Collaboration diagram for Certificate:



Public Types

- typedef std::unique_ptr< const Certificate, CustomDeleter > uptrc_t

Public Member Functions

- virtual std::uint32_t x509Version () const noexcept(true)=0
- virtual bool isRoot () const noexcept(true)=0
- virtual const X509DN & issuerDN () const =0
- virtual time_t startTime () const noexcept(true)=0
- virtual time_t endTime () const noexcept(true)=0
- virtual std::size_t serialNumber (WritableMemRegion ∗sn=nullptr) const noexcept(false)=0
- virtual std::size_t authorityKeyId (WritableMemRegion ∗id=nullptr) const noexcept(false)=0
- virtual std::size_t subjectKeyId (WritableMemRegion ∗id=nullptr) const noexcept(false)=0
- virtual bool verifyMe (const Certificate ∗caCert=nullptr) const noexcept(false)=0

9.76.1 Detailed Description

[SWS_CRYPTO_40200] Interface definition: ⌈ X.509 Certificate interface

⌋(*RS_CRYPTO_02306*)

9.76.2 Member Typedef Documentation

9.76.2.1 typedef std::unique_ptr<const Certificate, CustomDeleter> uptrc_t

[SWS_CRYPTO_40201] Interface definition: ⌈ Unique smart pointer of the interface.

⌋(*RS_CRYPTO_02404*)

9.76.3 Member Function Documentation

9.76.3.1 virtual std::uint32_t x509Version ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40211] Interface definition: ⌈ Get the X.509 version of this certificate object.

Returns

X.509 version

⌋(*RS_CRYPTO_02311*)

9.76.3.2 virtual bool isRoot ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40212] Interface definition: ⌈ Check whether this certificate belongs to a root CA.

Returns

`true` if this certificate belongs to a root CA (i.e. the certificate is self-signed).

⌋(*RS_CRYPTO_02311*)

9.76.3.3 virtual const X509DN& issuerDN ( ) const `[pure virtual]`

[SWS_CRYPTO_40213] Interface definition: ⌈ Get the issuer certificate DN.

Returns

Issuer DN of this certificate.

⌋(*RS_CRYPTO_02311*)

9.76.3.4 virtual time_t startTime ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40214] Interface definition: ⌈ Get the "Not Before" of the certificate.

Returns

"Not Before" of the certificate.

⌋(*RS_CRYPTO_02311*)

9.76.3.5 virtual time_t endTime ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40215] Interface definition: ⌈ Get the "Not After" of the certificate.

Returns

"Not After" of the certificate.

⌋(*RS_CRYPTO_02311*)

9.76.3.6 virtual std::size_t serialNumber (WritableMemRegion ∗ *sn* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40216] Interface definition: ⌈ Get the serial number of this certificate.

Parameters

`out`	*sn*	Optional pointer to an output buffer for storing the serial number.

Returns

Size of the certificate serial number in bytes.

Note

If `(sn == nullptr)` then this method only returns required size of the output buffer.

Exceptions

X509InvalidArgument	if `(sn != nullptr)`, but its size is not enough for storing the output value.

⌋*(RS_CRYPTO_02311)*

9.76.3.7 virtual std::size_t authorityKeyId (WritableMemRegion ∗ *id* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40217] Interface definition: ⌈ Get the DER encoded AuthorityKeyIdentifier of this certificate.

Parameters

`out`	*id*	An optional pointer to the output buffer.

Returns

Size of the DER encoded AuthorityKeyIdentifier in bytes.

Note

If `(id == nullptr)` then this method only returns required size of the output buffer.

Exceptions

CertInvalidArgument	if `(id != nullptr)`, but its size is not enough for storing the output value.

⌋*(RS_CRYPTO_02311)*

9.76.3.8 virtual std::size_t subjectKeyId (WritableMemRegion ∗ *id* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40218] Interface definition: ⌈ Get the DER encoded SubjectKeyIdentifier of this certificate.

Parameters

`out`	*id*	An optional pointer to the output buffer.

Returns

Size of the DER encoded SubjectKeyIdentifier in bytes.

Note

If `(id == nullptr)` then this method only returns required size of the output buffer.

Exceptions

CertInvalidArgument	if `(id != nullptr)`, but its size is not enough for storing the output value.

⌋(*RS_CRYPTO_02311*)

9.76.3.9 virtual bool verifyMe (const Certificate ∗ *caCert* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40219] Interface definition: ⌈ Verify signature of the certificate.

Parameters

`in`	*caCert*	An optional pointer to a Certification Authority certificate used for signature of the current one.

Returns

true if this certificate was verified successfully.

Note

Call with `(caCert == nullptr)` is applicable only if this is a certificate of a root CA.

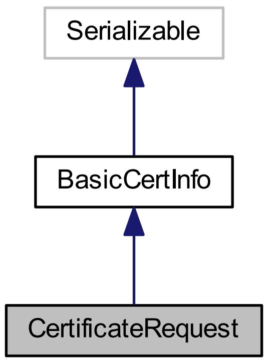
Exceptions

CertRuntimeError	if this or CA certificate has incorrect structure.

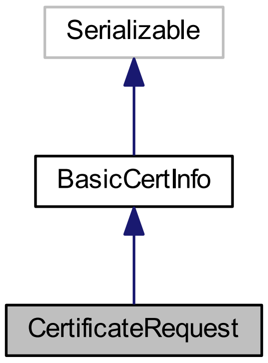
⌋(*RS_CRYPTO_02311*)

9.77 CertificateRequest Interface Reference

Inheritance diagram for CertificateRequest:



Collaboration diagram for CertificateRequest:



Public Types

- typedef std::unique_ptr< const CertificateRequest, CustomDeleter > uptrc_t

Public Member Functions

- virtual bool verify () const noexcept(true)=0
- virtual std::size_t challengePassword (std::string ∗password=nullptr) const noexcept(false)=0

9.77.1 Detailed Description

[SWS_CRYPTO_40300] Interface definition: ⌈ Certificate Request object interface

⌋(*RS_CRYPTO_02306*)

9.77.2 Member Typedef Documentation

9.77.2.1 typedef std::unique_ptr<const CertificateRequest, CustomDeleter> uptrc_t

[SWS_CRYPTO_40301] Interface definition: ⌈ Unique smart pointer of the interface.

⌋(*RS_CRYPTO_02404*)

9.77.3 Member Function Documentation

9.77.3.1 virtual bool verify ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40311] Interface definition: ⌈ Verifies self-signed signature of the certificate request.

Returns

`true` if the signature is correct.

⌋(*RS_CRYPTO_02311*)

9.77.3.2 virtual std::size_t challengePassword (std::string ∗ *password* = *nullptr*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40312] Interface definition: ⌈ Get the challenge password for this request (if it was included to the request).

Parameters

out	*password*	Optional pointer to an output string.

Returns

Length of the password if it was provided or 0 otherwise.

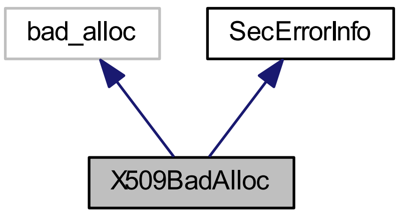
Exceptions

X509InvalidArgument	if `(password != nullptr)`, but its capacity is less then required for storing the password value.

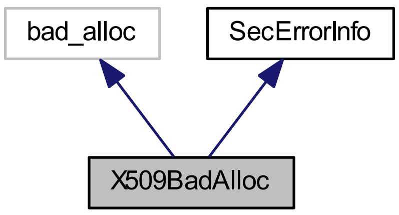
⌋(*RS_CRYPTO_02311*)

9.78 X509BadAlloc Interface Reference

Inheritance diagram for X509BadAlloc:



Collaboration diagram for X509BadAlloc:



Additional Inherited Members

9.78.1 Detailed Description

[SWS_CRYPTO_49500] Interface definition: ⌈ An interface for the X.509 Bad Allocation exception

⌋ *(RS_CRYPTO_02310)*

9.79 X509DN Interface Reference

Public Types

- typedef std::unique_ptr< const X509DN, CustomDeleter > uptrc_t

Public Member Functions

- virtual ∼X509DN ()=default
- virtual std::size_t getDN (std::string ∗dn=nullptr) const noexcept(false)=0
- virtual void setDN (const std::string &dn) noexcept(false)=0
- virtual std::size_t getAttribute (AttributeId id, std::string ∗attribute=nullptr) const noexcept(false)=0
- virtual void setAttribute (AttributeId id, const std::string &attribute) const noexcept(false)=0
- virtual std::size_t getAttribute (AttributeId id, unsigned index, std::string ∗attribute=nullptr) const noexcept(false)=0
- virtual void setAttribute (AttributeId id, unsigned index, const std::string &attribute) const noexcept(false)=0

9.79.1 Detailed Description

[SWS_CRYPTO_40400] Interface definition: ⌈ Interface of X.509 Distinguished Name (DN).

⌋ *(RS_CRYPTO_02306)*

9.79.2 Member Typedef Documentation

9.79.2.1 typedef std::unique_ptr<const X509DN, CustomDeleter> uptrc_t

[SWS_CRYPTO_40401] Interface definition: ⌈ Unique smart pointer of the interface.

⌋*(RS_CRYPTO_02404)*

9.79.3 Member Enumeration Documentation

9.79.3.1 enum AttributeId : std::uint8_t `[strong]`

[SWS_CRYPTO_40402] Interface definition: ⌈ Enumeration of DN attributes' identifiers.

⌋*(RS_CRYPTO_02311)*

Enumerator

COMMON_NAME	Common Name.
COUNTRY	Country.
STATE	State.
LOCALITY	Locality.
ORGANIZATION	Organization.
ORG_UNIT	Organization Unit.
STREET	Street.
POSTAL_CODE	Postal Code.
TITLE	Title.
SURNAME	Surname.
GIVEN_NAME	Given Name.
INITIALS	Initials.
PSEUDONYM	Pseudonym.
GENERATION_QUALIFIER	Generation Qualifier.
DOMAIN_COMPONENT	Domain Component.
DN_QUALIFIER	Distinguished Name Qualifier.
EMAIL	E-mail.
URI	URI.
DNS	DNS.
HOST_NAME	Host Name (UNSTRUCTUREDNAME)
IP_ADDRESS	IP Address (UNSTRUCTUREDADDRESS)
SERIAL_NUMBER	Serial Numbers.
USER_ID	User ID.

9.79.4 Constructor & Destructor Documentation

9.79.4.1 virtual ∼X509DN ( ) `[virtual]`, `[default]`

[SWS_CRYPTO_40410] Interface definition: ⌈ Destructor.

⌋(*RS_CRYPTO_02311*)

9.79.5 Member Function Documentation

9.79.5.1 virtual std::size_t getDN (std::string ∗ *dn* = `nullptr`) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40411] Interface definition: ⌈ Gets the whole Distinguished Name (DN) as a single string.

Parameters

`out`	*dn*	Pointer to a string for storing whole DN value as a single string.

Returns

Length of the whole DN string.

Note

Capacity of the output string must be enough for storing the output value!
If `(dn == nullptr)` then method only returns required buffer capacity.

Exceptions

CryptoInvalidArgument	if `(dn != nullptr)`, but `dn->capacity()` is less than required for the output storing!

⌋(*RS_CRYPTO_02311*)

9.79.5.2 virtual void setDN (const std::string & *dn*) `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40412] Interface definition: ⌈ Sets whole Distinguished Name (DN) from a single string.

Parameters

`in`	*dn*	A single string containing the whole DN value in text format.

Exceptions

CryptoInvalidArgument	if the dn string has incorrect syntax

⌋(*RS_CRYPTO_02311*)

9.79.5.3 virtual std::size_t getAttribute (AttributeId *id,* std::string ∗ *attribute = nullptr*) const [pure virtual], [noexcept]

[SWS_CRYPTO_40413] Interface definition: ⌈ Gets DN attribute by its ID (this method is applicale to all attributes except `ORG_UNIT` and `DOMAIN_COMPONENT`).

Parameters

`in`	*id*	The identifier of required attribute.
`out`	*attribute*	Pointer to a string for storing attribute value.

Returns

Length of the attribute value (0 for empty attributes).

Note

Capacity of the output string must be enough for storing the output value!
If `(attribute == nullptr)` then method only returns required buffer capacity.

Exceptions

CryptoInvalidArgument	if `(attribute != nullptr)`, but `attribute->capacity()` is less than required for the output storing!

⌋(*RS_CRYPTO_02311*)

9.79.5.4 virtual void setAttribute (AttributeId *id,* const std::string & *attribute*) const [pure virtual], [noexcept]

[SWS_CRYPTO_40414] Interface definition: ⌈ Sets DN attribute by its ID (this method is applicale to all attributes except `ORG_UNIT` and `DOMAIN_COMPONENT`).

Parameters

`in`	*id*	The identifier of required attribute.

in	*attribute*	A string containing the attribute value.

Exceptions

CryptoInvalidArgument	if the attribute string contains incorrect characters or has unsupported length.

⌋(*RS_CRYPTO_02311*)

9.79.5.5 virtual std::size_t getAttribute (AttributeId *id*, unsigned *index*, std::string ∗ *attribute* = `nullptr`) const `[pure virtual], [noexcept]`

[SWS_CRYPTO_40415] Interface definition: ⌈ Returns DN attribute by its ID and sequential index (this method is applicale to attributes ORG_UNIT and DOMAIN_COMPONENT).

Parameters

in	*id*	The identifier of required attribute.
in	*index*	The zero-based index of required component of the attribute.
out	*attribute*	Pointer to a string for storing attribute value.

Returns

Length of the attribute value (0 for empty attributes).

Note

Capacity of the output string must be enough for storing the output value!
If `(attribute == nullptr)` then method only returns required buffer capacity.

Exceptions

CryptoInvalidArgument	if `(attribute != nullptr)`, but `attribute->capacity()` is less than required for the output storing!
CryptoInvalidArgument	if `(id != ORG_UNIT) && (id != DOMAIN_COMPONENT) && (index > 0)`!

⌋(*RS_CRYPTO_02311*)

9.79.5.6 virtual void setAttribute (AttributeId *id*, unsigned *index*, const std::string & *attribute*) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40416] Interface definition: ⌈ Sets DN attribute by its ID and sequential index (this method is applicale to attributes `ORG_UNIT` and `DOMAIN_COMPONENT`).

Parameters

`in`	*id*	The identifier of required attribute.
`in`	*index*	The zero-based index of required component of the attribute.
`in`	*attribute*	A string containing the attribute value.

Exceptions

CryptoInvalidArgument	if the attribute string contains incorrect characters or has unsupported length.
CryptoInvalidArgument	if `(id != ORG_UNIT) && (id != DOMAIN_COMPONENT) && (index > 0)`!

⌋*(RS_CRYPTO_02311)*

9.80 X509Extensions Interface Reference

Public Types

- typedef std::unique_ptr< X509Extensions, CustomDeleter > sptr_t

Public Member Functions

- virtual ~X509Extensions ()=default
- virtual std::size_t count () const noexcept(true)=0

9.80.1 Detailed Description

[SWS_CRYPTO_40500] Interface definition: ⌈ Interface of X.509 Extensions.

⌋*(RS_CRYPTO_02306)*

9.80.2 Member Typedef Documentation

9.80.2.1 typedef std::unique_ptr<X509Extensions, CustomDeleter> sptr_t

[SWS_CRYPTO_40501] Interface definition: ⌈ Shared smart pointer of the interface.
⌋(*RS_CRYPTO_02404*)

9.80.3 Constructor & Destructor Documentation

9.80.3.1 virtual ~X509Extensions ( ) `[virtual]`, `[default]`

[SWS_CRYPTO_40510] Interface definition: ⌈ Destructor.
⌋(*RS_CRYPTO_02311*)

9.80.4 Member Function Documentation

9.80.4.1 virtual std::size_t count ( ) const `[pure virtual]`, `[noexcept]`

[SWS_CRYPTO_40511] Interface definition: ⌈ Gets number of elements in the sequence.

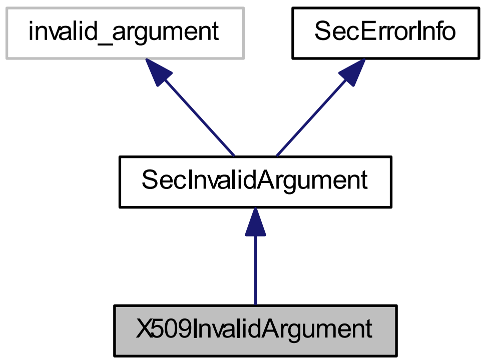
Returns

Number of elements in the sequence.

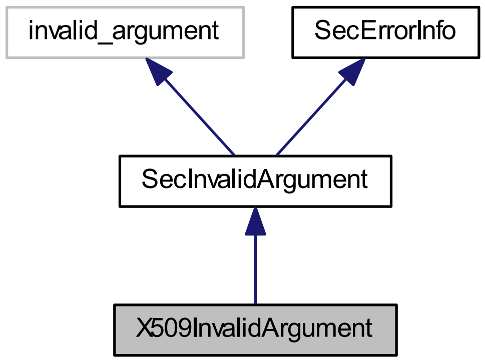
⌋(*RS_CRYPTO_02311*)

9.81 X509InvalidArgument Interface Reference

Inheritance diagram for X509InvalidArgument:



Collaboration diagram for X509InvalidArgument:



Additional Inherited Members

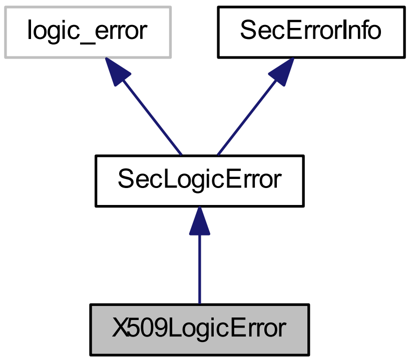
9.81.1 Detailed Description

[SWS_CRYPTO_49600] Interface definition: ⌈ An interface of the X.509 Invalid Argument exception

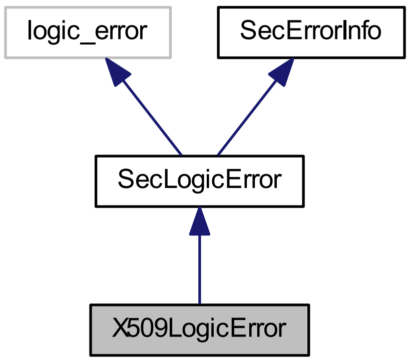
⌋*(RS_CRYPTO_02310)*

9.82 X509LogicError Interface Reference

Inheritance diagram for X509LogicError:



Collaboration diagram for X509LogicError:



Additional Inherited Members

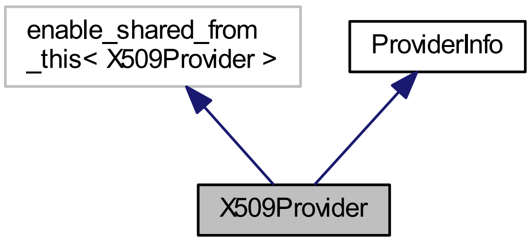
9.82.1 Detailed Description

[SWS_CRYPTO_49700] Interface definition: ⌈ A common interface for all X.509 Logic Error exceptions

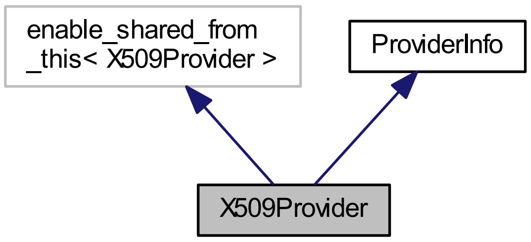
⌋*(RS_CRYPTO_02310)*

9.83 X509Provider Interface Reference

Inheritance diagram for X509Provider:



Collaboration diagram for X509Provider:



Public Types

- typedef std::shared_ptr< X509Provider > sptr_t

Additional Inherited Members

9.83.1 Detailed Description

[SWS_CRYPTO_40600] Interface definition: ⌈ X.509 Provider interface

Note

> All X.509 objects created by the provider should have an actual reference to their parent X.509 Provider.
> A X.509 Provider can be destroyed only after destroying of all its daughterly objects.
> Each method of this interface that creates a X.509 object is non-constant, because any such creation increases a references counter of the X.509 Provider.
> Any user of this interface should create shared pointers to it only by calls of the method `shared_from_this()`!

⌋(*RS_CRYPTO_02306*)

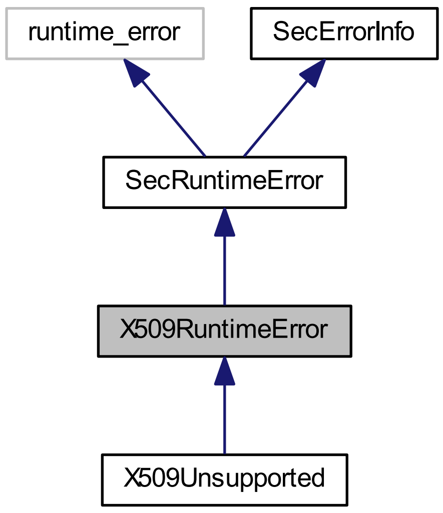
9.83.2 Member Typedef Documentation

9.83.2.1 typedef std::shared_ptr<X509Provider> sptr_t

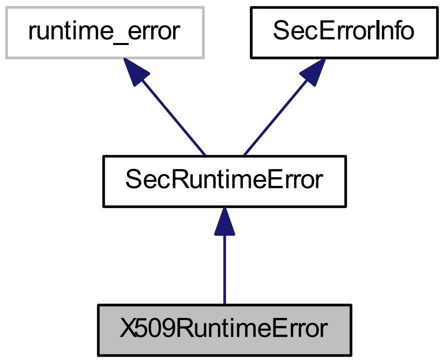
[SWS_CRYPTO_40601] Interface definition: ⌈ Shared smart pointer of the interface.
⌋(*RS_CRYPTO_02311*)

9.84 X509RuntimeError Interface Reference

Inheritance diagram for X509RuntimeError:



Collaboration diagram for X509RuntimeError:



Additional Inherited Members

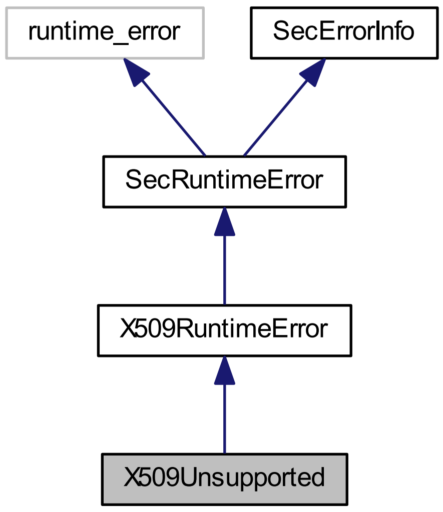
9.84.1 Detailed Description

[SWS_CRYPTO_49800] Interface definition: ⌈ A common interface for all X.509 Runtime Error exceptions

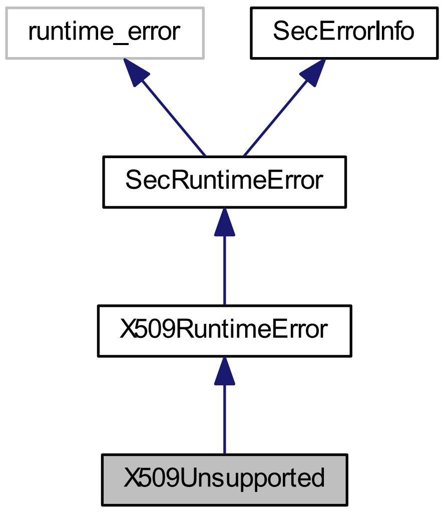
⌋*(RS_CRYPTO_02310)*

9.85 X509Unsupported Interface Reference

Inheritance diagram for X509Unsupported:



Collaboration diagram for X509Unsupported:



Additional Inherited Members

9.85.1 Detailed Description

[SWS_CRYPTO_49900] Interface definition: ⌈ An interface of the X.509 Unsupported method/argument exceptions

⌋*(RS_CRYPTO_02310)*

9.86 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

9.87 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

A Mentioned Class Tables

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document but which are not contained directly in the scope of describing specific meta model semantics.

Class	**ClientServerInterface**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Note	A client/server interface declares a number of operations that can be invoked on a server by a client. **Tags:** atp.recommendedPackage=PortInterfaces			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PortInterface*, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
operation	ClientServerO peration	1..*	aggr	ClientServerOperation(s) of this ClientServerInterface. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=blueprintDerivation Time
possibleErr or	ApplicationErro r	*	aggr	Application errors that are defined as part of this interface.

Table A.1: ClientServerInterface

Class	**CryptoJob**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Deployment::Crypto			
Note	This meta-class represents the ability to model a crypto job. The latter in turn represents a call to a specific routine that implements a crypto function and that uses a specific key and refers to a specific primitive as a formal representation of the crypto algorithm. **Tags:** atp.ManifestKind=MachineManifest; atp.Status=draft			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
cryptoKey	CryptoKeySlot	0..1	ref	This represents the key slots to which the referencing crypto job applies. **Tags:** atp.Status=draft
primitive	CryptoPrimitive	1	aggr	This aggregation defines the crypto primitive applicable for the enclosing crypto job. **Tags:** atp.Status=draft

Table A.2: CryptoJob

Class	**CryptoNeed**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			
Note	This meta-class represents a statement regarding the applicable crypto use case. **Tags:** atp.Status=draft; atp.recommendedPackage=CryptoNeeds			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	*Type*	*Mul.*	*Kind*	*Note*
primitiveFamily	String	1	attr	This attribute represents the ability to specify the algorithm family of the crypto need. **Tags:** atp.Status=draft

Table A.3: CryptoNeed

Class	***PortPrototype* (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Base class for the ports of an AUTOSAR software component. The aggregation of PortPrototypes is subject to variability with the purpose to support the conditional existence of ports.			
Base	*ARObject*, *AtpBlueprintable*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	*AbstractProvidedPortPrototype*, *AbstractRequiredPortPrototype*			
Attribute	*Type*	*Mul.*	*Kind*	*Note*
clientServerAnnotation	ClientServerAnnotation	*	aggr	Annotation of this PortPrototype with respect to client/server communication.
delegatedPortAnnotation	DelegatedPortAnnotation	0..1	aggr	Annotations on this delegated port.
ioHwAbstractionServerAnnotation	IoHwAbstractionServerAnnotation	*	aggr	Annotations on this IO Hardware Abstraction port.
modePortAnnotation	ModePortAnnotation	*	aggr	Annotations on this mode port.
nvDataPortAnnotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port.
parameterPortAnnotation	ParameterPortAnnotation	*	aggr	Annotations on this parameter port.
portPrototypeProps	PortPrototypeProps	0..1	aggr	This attribute allows for the definition of further qualification of the semantics of a PortPrototype. **Tags:** atp.Status=draft
senderReceiverAnnotation	SenderReceiverAnnotation	*	aggr	Collection of annotations of this ports sender/receiver communication.
triggerPortAnnotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port.

Table A.4: PortPrototype

Class	**RPortPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Component port requiring a certain port interface.			
Base	*ARObject*, *AbstractRequiredPortPrototype*, *AtpBlueprintable*, *AtpFeature*, *Atp Prototype*, *Identifiable*, *MultilanguageReferrable*, *PortPrototype*, *Referrable*			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
requiredInte rface	PortInterface	1	tref	The interface that this port requires, i.e. the port depends on another port providing the specified interface. **Stereotypes:** isOfType

Table A.5: RPortPrototype

Class	***Referrable* (abstract)**			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable			
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base	*ARObject*			
Subclasses	*AtpDefinition*, BswDistinguishedPartition, *BswModuleCallPoint*, BswModuleClient ServerEntry, BswVariableAccess, CouplingPortTrafficClassAssignment, Diagnostic DebounceAlgorithmProps, *DiagnosticEnvModeElement*, EthernetPriority Regeneration, EventHandler, ExclusiveAreaNestingOrder, *HwDescriptionEntity*, *ImplementationProps*, LinSlaveConfigIdent, ModeTransition, *Multilanguage Referrable*, PncMappingIdent, *SingleLanguageReferrable*, SocketConnectionBundle, SomeipRequiredEventGroup, TimeSyncServerConfiguration, TpConnectionIdent			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Tags:** xml.enforceMinMultiplicity=true; xml.sequenceOffset=-100
shortName Fragment	ShortNameFra gment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:** xml.sequenceOffset=-90

Table A.6: Referrable

B Span

This section shall elaborate the concept of the `span` introduced in [SWS_CRYPTO_01102]. The listing below illustrates the interface of the span.

```
class span {
public:
  ~span() = default;

```

```
// internal types (further types omitted for easy readability)
using pointer = element_type*;
using reference = element_type&;

// range access
/**
 * Retrieves the first \p count elements from the span.
 *
 * \param[in] count The number of elements to have in the subspan.
 * \return A subspan of \p count elements from the beginning.
*/
constexpr span<element_type, dynamic_extent> first(index_type count)
    const;
/**
 * Retrieves the last \p count elements from the span.
 *
 * \param[in] count The number of elements to have in the subspan.
 * \return A subspan of \p count elements from the end.
*/
constexpr span<element_type, dynamic_extent> last(index_type count)
    const;

/**
 * Retrieves a view on the span beginning from \p offset and
    containing \p count elemnents.
 *
 * \param[in] offset The index of the first element to be in the
    returned subspan.
 * \param[in] count The number of elements to have in the subspan.
 * \return A subspan of \p count elements from the element at \p
    offset.
*/
constexpr span<element_type, dynamic_extent> subspan(index_type
    offset, index_type count = dynamic_extent) const;

// size information
/**
 * Return the number of elements in the span
 *
 * \return The number of elements.
*/
constexpr index_type length() const;
/**
 * Return the number of elements in the span
 *
 * \return The number of elements.
*/
constexpr index_type size() const;
/**
 * Return the number of bytes in the span.
 *
 * \return The number of bytes.
*/
constexpr index_type length_bytes() const;
/**
 * Return the number of bytes in the span.
```

```
     *
     * \return The number of bytes.
    */
    constexpr index_type size_bytes() const;

    /**
     * Query if there are elements in the span.
     *
     * \return False if there are elements in the span, true otherwise.
    */
    constexpr bool empty() const;

    // element access
    /**
     * Access the element at \p idx.
     *
     * \param[in] idx The index where the element is located.
     * \return A reference to the element located an \p idx.
    */
    constexpr reference at(index_type idx) const;
    /**
     * Access the element at \p idx.
     *
     * \param[in] idx The index where the element is located.
     * \return A reference to the element located an \p idx.
    */
    constexpr reference operator[](index_type idx) const;

    // data access
    /**
     * Access the data of the span directly.
     *
     * \return A pointer to the data managed by the span.
    */
    constexpr pointer data() const;

    // iterators
    /**
     * Obtain an iterator pointing to the first element in the span.
     *
     * \return An iterator pointing to the first element in the span.
    */
    iterator begin() const;
    /**
     * Obtain an iterator pointing to the position after the last element
         in the span.
     *
     * \return An iterator pointing to the position after the last
        element in the span.
    */
    iterator end() const;

    /**
     * Obtain a constant iterator pointing to the first element in the
        span.
     *
     * \return A constant iterator pointing to the first element in the
       span.
    */
    const_iterator cbegin() const;
    /**
     * Obtain a constant iterator pointing to the position after the last
        element in the span.
     *
     * \return A constant iterator pointing to the position after the
       last element in the span.
    */
    const_iterator cend() const;

    /**
     * Obtain a reverse iterator pointing to the first element in the
       reversed span (i.e. the last element in the non-reversed span).
     *
     * \return A reverse iterator.
    */
    reverse_iterator rbegin() const;
    /**
     * Obtain a reverse iterator pointing to the position after the last
       element in the reversed span (i.e. before the fist element in
       the non-reversed span).
     *
     * \return A reverse iterator.
    */
    reverse_iterator rend() const;

    /**
     * Obtain a constant reverse iterator pointing to the first element
       in the reversed span (i.e. the last element in the non-reversed
       span).
     *
     * \return A constant reverse iterator.
    */
    const_reverse_iterator crbegin() const;
    /**
     * Obtain a constant reverse iterator pointing to the position after
       the last element in the reversed span (i.e. before the fist
       element in the non-reversed span).
     *
     * \return A constant reverse iterator.
    */
    const_reverse_iterator crend() const;
}
```